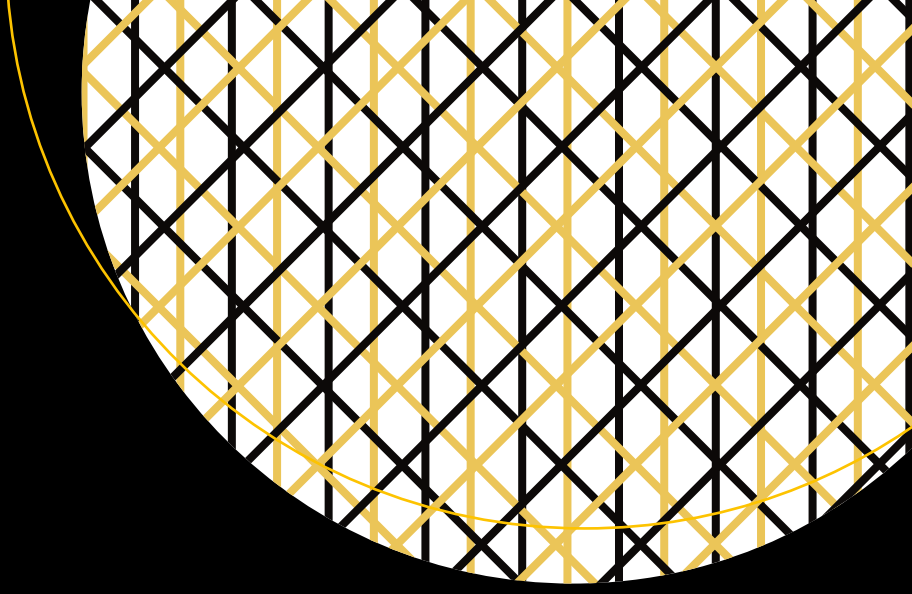


Database Benchmarking and Stress Testing

An Evidence-Based Approach to Decisions on Architecture and Technology

Bert Scalzo

Apress®

Database Benchmarking and Stress Testing

An Evidence-Based Approach to Decisions on Architecture and Technology

Bert Scalzo

Apress®

Database Benchmarking and Stress Testing

Bert Scalzo
Flower Mound, TX, USA

ISBN-13 (pbk): 978-1-4842-4007-6 ISBN-13 (electronic): 978-1-4842-4008-3
https://doi.org/10.1007/978-1-4842-4008-3

Library of Congress Control Number: 2018960192

Copyright © 2018 by Bert Scalzo

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484240076. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

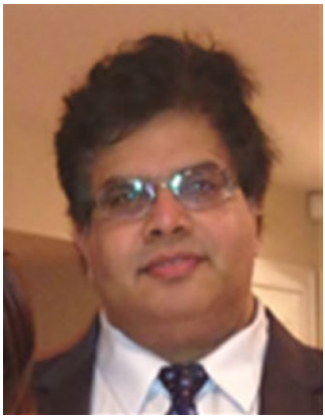
To my wife, Susan, who has put up with me for 30+ years
and our four legged children over the years:
Ziggy, Max, and Dexter.

Table of Contents

About the Author

Bert Scalzo is an Oracle ACE, blogger, author, speaker, and database technology consultant. He has a BS, MS, and PhD in computer science; an MBA; and has worked for over 30 years with all major relational databases, including Oracle, SQL Server, DB2 LUW, Sybase, MySQL, and PostgreSQL. Moreover Bert has also has worked for several of those database vendors. He has been a key contributor for many popular database tools used by millions of people worldwide, including TOAD, Toad Data Modeler, ERwin, ER/Studio, DBArtisan, Aqua Data Studio, and Benchmark Factory. In addition, Bert has presented at numerous database conferences and user groups, including SQL Saturday, SQL PAAS, Oracle Open World, DOUG, ODTUG, IOUG, OAUG, RMOUG, and many others. His areas of interest include data modeling, database benchmarking, database tuning and optimization, "star schema" data warehouses, Linux®, and VMware®. Bert has written for Oracle Technology Network (OTN), *Oracle Magazine*, *Oracle Informant*, *PC Week* (eWeek), *Dell Power Solutions Magazine*, *The LINUX Journal*, LINUX.com, Oracle FAQ, and Toad World. Moreover Bert has written an extensive collection of books on database topics, focusing mainly around TOAD, data warehousing, database benchmarking, and basic introductions to mainstream databases.

About the Technical Reviewer



Arup Nanda has been an Oracle database administrator (DBA) since 1993, dealing with everything from modeling to security, and he has a lot of gray hairs to prove it. He has coauthored 5 books, written 500+ published articles, presented 300+ sessions, delivered training sessions in 22 countries, and actively blogs at arup.blogspot.com. He is an Oracle ACE Director, a member of Oak Table Network, an editor for *SELECT Journal* (the IOUG publication), and a member of the Board for Exadata SIG. Oracle awarded him the DBA of the Year in 2003 and Architect of the Year in 2012. He lives in Danbury, CT, with his wife Anu and son Anish.

Acknowledgments

I have been doing database work, including benchmarking, for over three decades. Much of what I've learned has come from working alongside other great people who were very willing to share their expertise. So rather than list a few people and risk missing anyone, let me just say thanks to all those people “along the way” who’ve helped me to learn so much.

I also owe a debt of gratitude to the following companies for excellent opportunities and experience:

- IDERA
- Quest Software
- Embarcadero Technologies
- Logic Works
- Dell
- IBM
- EDS
- Citicorp
- Oracle Corporation
- Nationwide Insurance
- Battelle Memorial Institute

Introduction

Database Benchmarking and Stress Testing is for database administrators and professionals responsible for advising on any and all database architectural decisions ultimately affecting overall database performance. This book provides an empirical method for answering "what if" questions about database performance, helping database administrators make sound architectural decisions in a fast-changing landscape of virtualized servers and container-based solutions. Today's database administrators face numerous such questions:

- What if we consolidate databases using multitenant features?
- What if we virtualize database servers as Docker containers?
- What if we deploy the latest in NVMe flash disks to speed up IO access?
- Do features such as compression, partitioning, and in-memory OLTP earn back their price?
- What if we move our databases to the cloud?

As an administrator, do you know the answers or even how to test the assumptions? You should know how to test and find answers to these questions, and doing so is what this book is about. *Database Benchmarking and Stress Testing* introduces you to database benchmarking using industry-standard test suites such as the TCP series of benchmarks, which are the same benchmarks that vendors rely upon. You'll learn to run these industry-standard benchmarks and collect results to use in answering

questions about the performance impact of architectural changes, technology changes, even down to the brand of database software. You'll learn to measure performance and predict the specific impact of changes to your environment. You'll learn the limitations of the benchmarks and the crucial difference between benchmarking and workload capture/reply.

Database Benchmarking and Stress Testing is about creating empirical evidence in support of business and technology decisions. It's about not guessing when you should be measuring. Empirical testing is scientific testing that delivers measurable results. Begin with a hypothesis about the impact of a possible architecture or technology change. Then run the appropriate benchmarks to gather data and predict whether the change you're exploring will be beneficial, and by what order of magnitude. Stop guessing. Start measuring. Let *Database Benchmarking and Stress* Testing show the way.

CHAPTER 1

Benchmarking Basics

This book intends to enlighten readers on all things related to database benchmarking, stress testing, and workload capture/replay. At this moment you may feel that these are one and the same topic, or simply that what differentiates them is inconsequential. Moreover you may consider these topics as somewhat esoteric and thus more suited for academia and theoreticians, but not really relevant in the real world where you work. Finally, you might simply believe that the time and effort for performing such testing activities is unwarranted, or at least very difficult to justify. As a result, many people are skeptical about the true value of these activities. This book endeavors to transform that mistaken opinion or at least enlighten the reader as to the genuine value of these topics.

This first chapter provides all of the basic required definitions and our common vocabulary for database benchmarking, stress testing, and workload capture/replay. Moreover it explains the why, when, and how these different techniques are worthwhile. Finally, and most importantly, this chapter expounds upon how to best choose from among these different methods, because we all generally find that knowing when to use a specific tool for a particular job in life is what is most valuable. When we see a task that involves a nail we think hammer, when it involves a screw we think screwdriver. Once you have that same kind of instinctually "*gut feel*" for these database techniques, you'll find each to be an invaluable tool on your tool belt. It will take some time to reach this level of comfort, but it is well worth it.

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_1

What Is a Database Stress Test?

This may seem like a trivially easy question, but the answer may be more significant than you thought. Most people have a basic knowledge of the simple procedure known as a cardiac stress test. That's the procedure where a doctor, most often a cardiologist, hooks you up to a heart monitor and either has you walk briskly or run on a treadmill for some extended duration. The goal in lay terms is to "*get your heart rate up*" and "*to make you sweat.*" The doctor of course is looking for more valuable detailed information by connecting the patient to an electrocardiogram (ECG) monitor. While such a test usually cannot identify a specific medical condition, it nonetheless has great predictive value for detection of early signs of conditions that warrant concern. In short, this simple medical test is invaluable, hence why it is so often performed.

Now what about your database, just how healthy is it? While your database performance under today's typical, well-known workload might not be a problem, what would happen if you pushed your database to "*get its heart rate up*" and "*to make it sweat*"? You might consider such database stress testing to fall under the simple and prudent category of capacity planning. That would allow the database administrator to answer critical management questions such as when will the database application's resource consumption increase to the point where ordering more computing resources is required? However useful as that may sound, it is not the usual reason you may be asked to or think it wise to perform any database stress testing (more about that later in this chapter).

So just how do you stress a database in order to "*get its heart rate up*" and "*to make it sweat*"? Unlike the cardiologist where the prescribed best practice and procedure are well defined and almost universally accepted, just how does one perform a reliable and repeatable stress test upon a database when all databases are essentially quite different? For many database administrators the answer is that they monitor the database during peak usage periods and capture an expectantly significant subset

of the database application's overall statements that seem expensive. These statements are then cobbled together into a single script or program that the database administrator can then execute to measure before and after results for a specific set of changes or over a period of time to gauge potential differences.

While seemingly quite reasonable, this simple approach has several shortcomings. First, it is very random in nature. What if the database administrator picks the wrong time period to monitor or misses some important and expensive database statement executions while monitoring? Second, once captured it becomes a snapshot of what the database application was on the day it was captured. What if the next version of the application changes significantly in nature? Third, different database administrators might not choose the same time periods to monitor or set database statement executions as important. In fact, the same database administrator might not be able to do so if repeating that monitor and capture process a second time. As a result this approach lacks practical reliability and predictability. It does have some limited value, but cannot realistically serve as the generally accepted method for first measuring and then comparing database performance. Something more is needed.

What Is a Database Benchmark?

Generally speaking a database benchmark is a well-defined and universally accepted, industry standard database stress test with clear performance metrics to measure and compare results for a given database size and concurrent user workload. Moreover these stress tests are regarded as principally reliable across repeat runs given the same parameters. That means the performance scores attributed should not vary significantly across test runs, statistically speaking. Moreover such stress tests are considered both viable and reliable across different

database versions, database vendors, and even significantly different database deployment scenarios (i.e., upon dedicated vs. virtual server, on premise vs. in the cloud, etc.). In short, a database benchmark is the database administrator's logical equivalent to the cardiologist's treadmill heart stress test.

So just where do these database benchmarks come from? Plus importantly, which of the various sources are considered to have created sincere and enduring industry standards? Finally, which sources and thus database benchmarks should a database administrator add to his or her tool belt? These are important questions that require a brief historical recount in order to provide proper background and context for meaningful appreciation.

With the emergence of Structured Query Language (SQL)-based relational databases in the early 1980s, it became far more important to be able to ascribe some performance metrics to new databases. Earlier generation network and hierarchical databases were generally highly performant due to their fundamental nature that was based on physical pointers. However newer relational databases based upon Codd's 12 rules were radically different, instead relying on logical references based upon the data values. This approach freed database programmers from focusing on "*how to access*" permitting them to focus on "*what to access.*" The database systems now became responsible for the data structures for storing the data and retrieval procedures for resolving queries. This fundamental shift from physical to logical data access was not cost free, so the need arose to be able to reliably measure that cost, hence the new need for database benchmarks.

An obvious early source of database benchmarks was academia since relational databases were so novel and based upon the mathematics of relational algebra and relational calculus. One such early example was the

"*Wisconsin Benchmark*" in 1983[1] from the computer sciences department at the University of Wisconsin.

Its primary objective was to provide a method for testing the performance of all the major underlying components of a relational database system. Then in 1985 the "*Debit-Credit Benchmark,*"[2] later known as the TP1, debuted, written by Jim Gray with contributions from academia, database vendors, and even users. Then in 1987 Jim Gray authored the "*Top Gun Benchmark*"[3] using the Debit-Credit workload. From 1985 (after a database benchmarking article in *Datamation*) to 1988 a database benchmarking war ensued where hardware and software vendors began battles of one-upmanship. Gray recruited others and attempted to bring order to this chaos by forming the Transaction Processing Council (`www.tpc.org`), whose mission statement[4] reads:

> *The TPC is a non-profit corporation founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry.*

Another source of database benchmarks has been larger hardware vendors (e.g., Jim Gray was at Tandem Computers while creating some of his works). Some of these have been reasonably database agnostic. A good example is that Dell offers the very popular, open source "*DVD Store Benchmark*" for SQL Server, Oracle, MySQL, and PostgreSQL. This benchmark's purpose reads:

---

[1]Bitton, Dina; DeWitt, David, J.; Turbyfill, Carolyn; "Benchmarking Database Systems: A Systematic Approach," University of Wisconsin, Department of Computer Sciences, 1983, `http://pages.cs.wisc.edu/~dewitt/includes/benchmarking/vldb83.pdf`

[2]Gray, Jim; Et A; "A Measure of Transaction Processing Power," Tandem Computers Inc., 1985, `http://www.hpl.hp.com/techreports/tandem/TR-85.2.pdf`

[3]Jim Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann, San Mateo, CA, 1991.

[4]`www.tpc.org` website, "About the TPC: Mission Statement," `http://www.tpc.org/information/about/abouttpc.asp`

> *The DVD Store Version 2 (DS2) is a complete online e-commerce test application, with a backend database component, a web application layer, and driver programs. The goal in designing the database component as well as the mid-tier application was to utilize many advanced database features (transactions, stored procedures, triggers, referential integrity) while keeping the database easy to install and understand. The DS2 workload may be used to test databases or as a stress tool for any purpose.*

However while some of the hardware vendors' database benchmarks can be leveraged across database vendors like the Dell DVD Store benchmark with some degree of fairness and reliability, other hardware vendor database benchmarks whose focus is on specific components such as IO subsystems often cannot. A good example would be the extremely popular "*Silly Little Oracle Benchmark*" (SLOB). This benchmark was written to highlight raw database write performance on flash disk-based systems. As such, it designed more to test write activity than reads. Essentially it issues just two commands: large updates and select counts. Thus it really does not test the query optimizer or any other advanced aspect of the database engine – just how fast can it write data. That does not mean that it's a bad test or to avoid it. But you need to know what it does test and what it really reports. In fact, the author[5] clearly states this on his web page:

> *SLOB is not a database benchmark. SLOB is an Oracle I/O workload generation tool kit. I need to point out that by force of habit many SLOB users refer to SLOB with terms like benchmark and workload interchangeably. SLOB aims to fill the gap between Orion and CALIBRATE_IO (neither generate legitimate database I/O as explained partly here) and full-function transactional benchmarks (such as Swingbench). Transactional workloads are intended to test the transactional capability of a database.*

[5]Closson, Kevin; "Introducing SLOB – The Simple Database I/O Testing Toolkit for Oracle Database," https://kevinclosson.net/2012/02/06/introducing-slob-the-silly-little-oracle-benchmark/

Another source of database benchmarks has been some of the open source efforts, specifically some of the larger open source database projects. For example, PostgreSQL offers "*pgbench*" as the standard tool for comparing performance. While it makes running a test fairly easy, it only works with PostgreSQL and more importantly the documentation clearly says:

> *To benchmark the production application tests must be performed that are similar to production loads and query types. The pgbench queries for example do not perform joins, where clauses using regular expressions on text fields and other query types that would be found in many applications.*

In addition MySQL offers the "*MySQL Benchmark Tool*" that runs three database benchmarks for MySQL databases only:

1. DBT2
2. SysBench
3. flexAsynch

Where the documentation defines those database benchmarks as follows:

> *DBT2 is an open source benchmark that mimics an OLTP application for a company owning large amounts of warehouses. It contains transactions to handle New Orders, Order Entry, Order Status, Payment and Stock handling. FlexAsynch is a benchmark specifically developed to test scalability of MySQL Cluster. Sysbench is a popular open source benchmark to test open source DBMSs.*

It should be fairly clear by now that database benchmarking has a long, varied, and very interesting history. However it should also be evident that only the TPC benchmarks are designed to objectively test real-world database performance with fair results. We'll more fully cover all the relevant TPC database benchmarks in Chapter 2.

What Is Workload Capture/Replay?

For many people the concept of an industry standard database benchmark for a specific defined purpose such as "*online transaction processing*" (OLTP) or "*data warehousing*" (DW) is sufficient. But for some people such pre-canned "*synthetic*" tests are not. They prefer something more realistic and meaningful for their database application's actual nature. What they desire is essentially the first method of doing a database stress test (i.e., monitor and capture the commands), but for 100% of the activity and not just a manual, somewhat random sample. They want it all. They want a complete snapshot of all their database activity for a defined period of time. Furthermore they want it packaged in a way such that it can be replayed on demand. In a nutshell they want what is referred to as workload capture and replay. It is pretty much exactly what the name says, which for many database administrators is the only benchmark they consider meaningful.

It's hard to argue against such a natural and logical viewpoint. It simply makes good common sense. But as the proverb says, "*The road to hell is paved with good intentions*," meaning that good intentions when acted upon may have unintended consequences. For workload capture/replay to be truly representative you must use a database of exactly the same size and with the same data (or at least data skew) such that the results are accurate. That means copying your entire 100-Terabyte production database to test with the data intact as is, so no data subsetting or data masking. That translates into very large hardware and software costs, plus it poses a huge security risk as real production data is now completely available in your test environment. Your management may have a minor fit on the budget impact, but your security and compliance people will have your job for doing that.

Another major problem with workload capture and replay is that not every database vendor offers it, and for those who do the feature can be costly (the notable exception being Microsoft SQL Server). Plus of course

database vendors workload capture/replay tools only work for their database. For many shops, that would mean adopting multiple tools since we all deploy more than just one database vendor these days. There are also some third-party software vendors, but the database vendors do not generally share any of their system internals, so some people question just how effective and efficient such solutions can be without such access and insight. Nonetheless workload capture/replay remains the Holy Grail for those who want their database benchmark to reflect what their database application really performs.

Why These Techniques Are Useful

As was pointed out earlier, database benchmarking has its share of genuine skeptics. However none of us drives our vehicles the same way the environmental protection agency (EPA) does in order to verify the fuel economy, yet we all look for that window sticker posted on any new car we purchase, often treating that number as vital information when oil is expensive. So we clearly see the predictive value in that benchmark and accept it for what insights it provides. While we don't expect to achieve that same fuel economy rating in our driving, we nonetheless do genuinely expect it to be relatively accurate for comparing vehicles. As humans, having such a comparison with some belief as to its reliability is very reassuring, and makes the decision process a little less stressful.

There are literally dozens of such benchmarks people rely upon in everyday life, yet often database benchmarking is still treated with great skepticism. So let's examine some questions that you might be asked to answer for which you may need some insight that a benchmark might provide (assuming of course that an appropriate benchmark is selected). The point being that if you readily rely on an EPA gas mileage benchmark when buying a new car, then you must believe in the value of some benchmarks, so why not with your databases too?

When These Techniques Are Valuable

Imagine as the database administrator your boss asks you, "What impact can we expect on our production database performance if we were to":

- Apply a major version upgrade to the operating system and/or database?
- Migrate from proprietary UNIX on RISC CPUs to Linux on Intel CISC CPUs?
- Choose among equivalently priced servers from Dell, HP, and Lenovo?
- Migrate from a Windows Server to Linux, or vice versa?
- Migrate from raw iron to hypervisor?
- Migrate from commercial hypervisor like VMware to open source such as KVM?
- Consolidate databases per instance?
- Migrate from on premise to the cloud?
- Upgrade to the next cloud pricing tier?

These are but just a few such questions. So how would you answer questions like these? Even with 30+ years of database experience, I could only reasonably provide my opinion on a few of these, and probably not for the contextual specifics in play required to provide management with reliable recommendations. So in such cases the experienced database administrator will perform some testing, often labeled as a "*proof of concept*" (POC) project. It's at that time when you must decide whether you will perform just simple stress testing, database benchmarking, or workload capture/replay.

Now a logical pushback is often, "We don't make those types of decisions very often, so it's not too important or worth worrying about until and when we have such questions." I really do get it, database administrators are forever being asked and/or told to do more with less. They simply do not have sufficient time or energy to master every little technique. But if anything, the past decade has shown databases is that where databases are deployed has evolved, first being virtualized and are now moving to the cloud. So these types of questions and issues are happening far more than in days past. In fact, it's rare these days that database administrators or data architects don't fact such questions.

How to Choose Between These Options

So now that we've examined both why and when these various techniques are of importance, the remaining question is which one should you chose? As with any big decision in life, the legitimate and unflippant answer is "*It all depends.*" However I advocate that in many if not most cases, you should give database benchmarking practical consideration. It is arguably quicker and more reliable than creating a stress test script via ad hoc monitoring the database to capture what you believe and hope is a truly representative workload. It also does not require using real data that may be confidential and possibly even illegal to copy into a test environment. Finally it does not add the extra cost of a workload capture/replay tool and it too relies upon using real data.

Yes you could argue that database benchmarking relies upon a synthetic workload. However, numerous database benchmarks exist that approximate all of the various workload types, including online transaction processing (OLTP); data warehousing (DW) queries; batch processing systems; and user defined, weighted combinations of all the above. So you really should be able to quickly and easily cobble together a mixture of benchmark tests that meaningfully and acceptably approximate

your system. This will permit you to then focus on quickly and accurately comparing "*apples to apples*" scenarios.

As for veteran database administrators, you may already have a feel for what you would do or possibly even a preference among the proposed choices. I will simply suggest that you read this book with a fair and open mind that database benchmarking might very well augment your existing repertoire. Remember what the Chinese philosopher Confucius said, "You cannot open a book without learning something."

As for more junior database administrators or those simply unaccustomed to such tough "*what if*" database performance questions, you may hope this book provides all the answers. However no book could ever possess all of your experience and insights into your database and its data. Thus I suggest that you read this book looking for a best fit of your goals vs. the database context you're working in. As Bill Nye the "*Science Guy*" said, "Everyone you will ever meet knows something you don't, and therefore you should listen to others and "respect their knowledge."

Where We Go from Here

Assuming that you now are far more accepting of the idea of database benchmarking, you might well ask the following questions:

- **Chapter 2**
  - What industry standard database benchmarks should I learn and use?
- **Chapter 3**
  - What software tools exist to support varying database benchmarking efforts?
- **Chapter 4**
  - How does one prepare for a real-world database benchmarking effort?

- **Chapter 5**
  - What common problems or missteps occur during a database benchmarking effort?
- **Chapter 6**
  - What hardware options are important during a database benchmarking effort?
- **Chapter 7**
  - What software options are important during a database benchmarking effort?
- **Chapter 8**
  - How does one effectively benchmark for database consolidation?
- **Chapter 9**
  - How does one effectively benchmark for database virtualization?
- **Chapter 10**
  - How does one effectively benchmark for databases in the public cloud?
- **Chapter 11**
  - How does one effectively perform a database workload capture and replay?
- **Chapter 12**
  - How does one effectively perform a simple database stress test?

CHAPTER 2

Industry Standard Benchmarks

In Chapter 1 we differentiated between a database benchmark, database stress test, and workload capture/replay. By understanding these differing approaches, we formed the foundation for a common understanding and vocabulary. Plus we learned why these different techniques are useful and when to best utilize them. To recap, our basic definitions were:

- A database stress test is an indefinite and limited in nature, artificial workload meant to basically stress the database's low-level performance (i.e., to make the database sweat).
- A database benchmark is a well-defined and universally accepted, industry standard database stress test with clear performance metrics to measure and compare results for a given database size and concurrent user workload.
- A workload capture is a well-defined and limited snapshot of actual database application activity recorded to create a database benchmark with a non-artificial workload to be replayed to compare before and after performance.

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_2

In this chapter we'll now review all of the industry standard database benchmarks of significance. Moreover, we'll break them down into two major categories: those which are current and those now out of favor. Most of these database benchmarks were developed by the Transaction Processing Council (www.tpc.org) while a few originated at universities. It's important to know both the current and unpopular industry standard database benchmarks as the former often have roots in the latter. Finally note that each industry standard database benchmark tends to focus on a particular type of workload; therefore knowing all these benchmarks will enable you to select the proper one for your needs. In fact, you may well discover that you require a mixture of benchmarks in order to best approximate the workload you desire to test. So once again the knowledge of all these benchmarks will be critical in your pursuit of testing.

Most Popular Benchmarks

Let us now discuss the most popular benchmarks.

TPC-C

The TPC-C is probably the most recognized, least understood, and often misquoted database benchmark. It's also the one offered by the most database benchmarking tools, but which may not adhere to the intent of the benchmark's specification due to program defaults or user definable options that are out of spec. Moreover, these database benchmarking tools may ask for user supplied runtime parameters that define the nature of the test but which users do not know how to correctly provide unless they have read the database benchmark spec. Furthermore, most database benchmarking tools do not report their results in a consistent manner or achieve radically different results based upon their default settings (which may or may not be to spec). Finally, most people focus on just a portion of

the test results in a vacuum, such as transactions per second (TPS, without the proper context for any meaningful interpretation. As such the TPC-C has garnered a bad reputation for all the wrong reasons.

Let's start by looking at the TPC-C database benchmark definitions from the tpc.org web site and the TPC-C spec:

- Website: TPC-C is an OLTP benchmark. It's more complex than previous OLTP benchmarks such as TPC-A because of its multiple transaction types, more complex database, and overall execution structure.
- TPC-C spec: TPC-C is an OLTP workload. It's a mix of read-only and update intensive database transactions that simulate the activities of a complex OLTP application.

Neither definition really provides any kind of meaningful insight into the nature of the application workload other than saying it is OLTP based. Here is a more thorough and therefore more useful definition:

> *Approved in 1992 the TPC-C is the original OLTP benchmark whose workload of five transactions (New-Order, Payment, Order-Status, Delivery and Stock-Level) attempts to approximate a wholesale distributor with a fairly small number of warehouses to service a relatively large number of retail locations. Performance is a measure of new orders per minute.*

Note that the TPC-C is now over 25 years old. It does not adequately mimic today's real-world database workloads, nor does it properly stress the capabilities of today's hardware and database engines. As such, the TPC-C is quickly losing favor. The TPC-C is now considered awfully "*old school*" and has been superseded by the TPC-E, a newer and far more complex OLTP database benchmark covered later in this chapter. Nonetheless the TPC-C remains the most widely used and heavily quoted benchmark, hence why it's critical to cover TPC-C database benchmark in

great detail. Let's continue our investigation by examining the TPC-C data model shown in Figure 2-1.

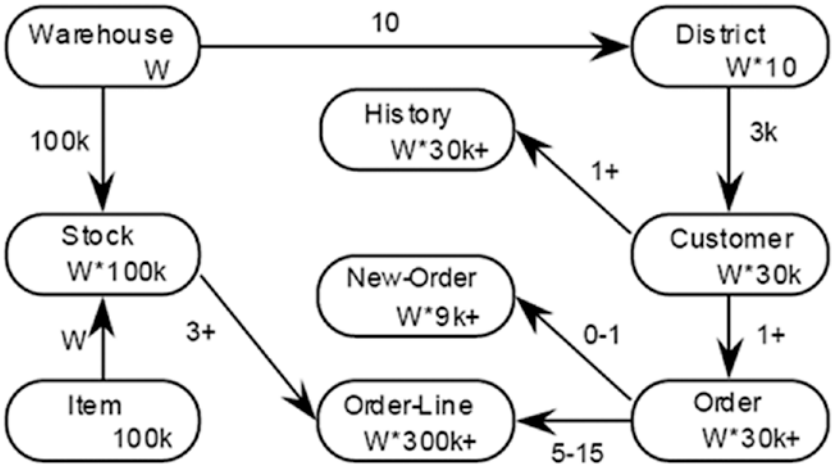


Figure 2-1. *TPC-C Data Model*

First note that this data model is by today's typical database application standards quite simple and trivially small with just nine tables. In fact, it's seen as so simple and small to arguably be of little or no value, hence why the newer TPC-E database benchmark was created. Second observe that the design begins in the upper left corner with the WAREHOUSE table that possesses a 10 to 1 parent/child relationship with the DISTRICT table. The key idea being that each WAREHOUSE possesses 10 terminals, which equates to a concurrent user session possible. This is critical as the spec clearly mandates a database size or scale factor of 1 warehouse per 10 concurrent database user sessions. If you try to run a high user workload with an out of spec low WAREHOUSE count such as 100 users with a scale factor of 1, then your benchmark results will likely be skewed and you may well encounter database contention or locking issues. Yet most database benchmarking tools permit their users to do just that, that is, request running 1,000 concurrent database sessions with a scale factor far less than the 100 required by the spec.

To better understand this, we shall inspect the logical database design mandated by the spec where you will find that the DISTIRCT table has a built-in inefficiency. Examine the required table definition for the DISTRICT table shown in Figure 2-2.

DISTRICT Table Layout

Field Name	Field Definition	Comments
D_ID	20 unique IDs	*10 are populated per warehouse*
D_W_ID	2*W unique IDs	
D_NAME	variable text, size 10	
D_STREET_1	variable text, size 20	
D_STREET_2	variable text, size 20	
D_CITY	variable text, size 20	
D_STATE	fixed text, size 2	
D_ZIP	fixed text, size 9	
D_TAX	signed numeric(4,4)	*Sales tax*
D_YTD	signed numeric(12,2)	*Year to date balance*
D_NEXT_O_ID	10,000,000 unique IDs	*Next available Order number*

Primary Key: (D_W_ID, D_ID)

D_W_ID Foreign Key, references W_ID

Figure 2-2. *DISTRICT Table Layout*

Note that the D_NEXT_O_ID column holds the next available order number, so a session must lock the row in order to fetch the current value and then increment it. This is the old-fashioned way to implement this concept before auto increment columns or table triggers tied to sequence number generators became available. It is done this way since not all databases offer these advanced features. That's why running the wrong user load for given scale factor such as 1,000 users with a scale factor of 1 ends up with severe database contention and locking issues as all the sessions must fight for locks on the limited number of DISTRICT

table rows. Yet I almost universally see this mistake by people who've not read the benchmark spec and who just complain that the database benchmarking tools stink (i.e., won't report the extremely high numbers they expect and want to see).

Another aspect of the database design is what advanced physical database options or schema object definition modifications are permissible? The TPC-C spec is fairly clear about what can and cannot be done. The following are allowed:

- Unique indexes for primary key constraints
- Horizontal table partitioning
- Vertical table partitioning
- Replication of tables

One advanced database feature not permitted is physical table data row clustering; however you will find many published benchmark test results that utilize this feature and are therefore non-sanctioned. Fortunately, most database benchmarking tools generally do not offer table clustering either by default or as an option. Some, however, do offer to partition tables horizontally. But often the partitioning scheme is rather generic and can quite often be outdone by the DBA manually defining the partitioning scheme to best leverage their hardware and database optimizer. Finally note that as an older database benchmark from times when databases had far fewer features, the TPC-C does not recommend nor require referential integrity (i.e., foreign key constraints) or indexes upon them. However, as this is a modern best practice, most database benchmarking tools will create foreign keys and indexes on those FK columns. While this is technically speaking "*out of spec*," it seems like a fair improvement if utilized consistently. Plus it just "*feels right*" when working with a relational database.

TPC-E

The TPC-E is a modern OLTP test meant to replace the TPC-C. The TPC-E attempts to resolve all the known shortcomings of the TPC-C. Since it first debuted circa 2007, it is far less known and not yet as popular except with some of the hardware and database vendors. While database administrators may have heard of the TPC-E, hardly any have run this database benchmark. Furthermore, few database benchmarking tools implement the TPC-E; in fact only two currently do so this will be covered in Chapter 3 where we review the popular database benchmarking software currently available.

As before, let's start by looking at the TPC-E database benchmark definition from the tpc.org web site and the TPC-E spec:

- Website: TPC-E is more complex than previous OLTP benchmarks such as TPC-C because of its diverse transaction types, more complex database, and overall execution structure. TPC-E involves a mix of 12 concurrent transactions of different types and complexity.
- TPC-E spec: TPC-E is an OLTP workload that is a mixture of read-only and update-intensive transactions that simulate the activities found in modern, complex OLTP application environments.

Once again neither definition really provides any kind of meaningful insight into the nature of the application workload other than saying it's a more complex and modern OLTP benchmark. Here is a more thorough and therefore more useful definition:

> *The TPC-E benchmark simulates the OLTP workload of a brokerage firm. The focus of the benchmark is the central database that executes transactions related to the firm's customer accounts. Although the underlying business model of TPC-E is a brokerage firm, the database schema, data population, transactions, and implementation rules have been designed to be broadly representative of modern OLTP systems.*

The TPC-E design really does appear, behave, and feel like a genuine database application that an internet-based brokerage house might implement. Most interestingly the TPC-E contains both Consumer-to-Business as well as Business-to-Business type transactions. Furthermore, four characteristics from the spec that contribute to its highly realistic nature are:

- The simultaneous execution of multiple transaction types that span a breadth of complexity
- A balanced mixture of disk input/output and processor usage
- A mixture of uniform and non-uniform data access through primary and secondary keys
- Databases consisting of many tables with a wide variety of sizes, attributes, and relationships with realistic content

As before, let's continue our investigation by examining the TPC-E data model shown in Figure 2-3.

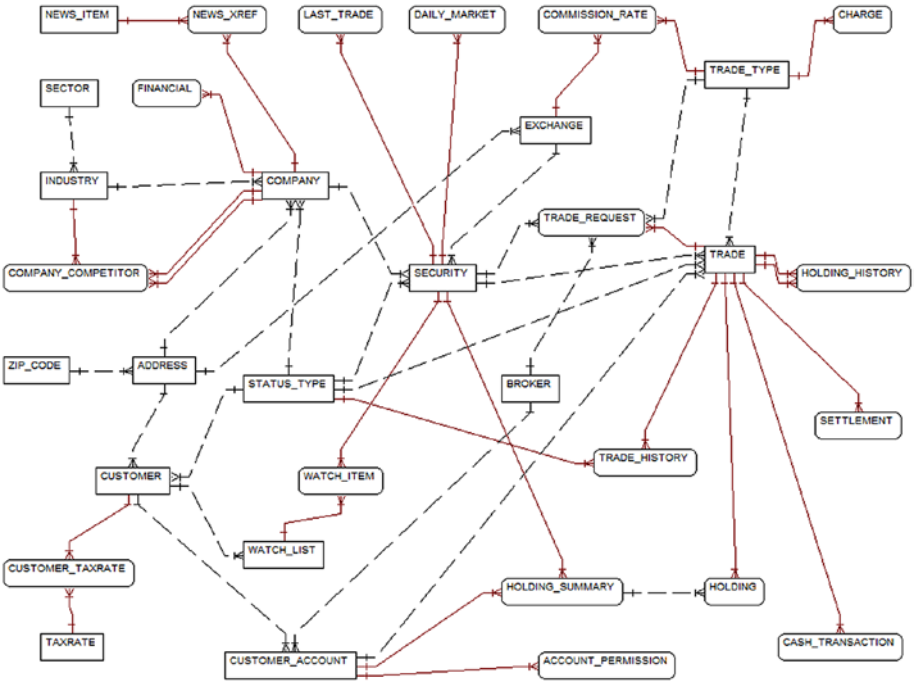


Figure 2-3. *TPC-E Data Model*

It's not important to examine every table shown in Figure 2-3 in detail. What's important to note is that it looks more like a real-world data model and not just one from some academic exercise. Unlike the TPC-C that has few tables and whose relationships are simple parent/child in nature, the TPC-E has many tables and also complex relationships. In fact, there are several important differences between these two benchmarks as highlighted in Table 2-1 shown below.

Table 2-1. *Comparison of TPC-C vs. TPC-E*

Characteristic	TPC-C	TPC-E
Business Type	wholesale distributor	brokerage house
Total # Tables	9	33
Total # Columns	92	188
Cols/Table Range	3 – 21	2 - 24
Avg Cols/Table	10.2	5.7
# RO Transactions	2	6
% RO Transactions	8%	76.9%
# RW Transactions	3	4
% RW Transactions	92%	23.1%
Data generation	random	census data
Check Constraints	0	22
Referential Integrity	NO	YES

There are four key differences to note from Table 2-1. First, the total number of tables in the TPC-E is 3.67 times as large as that of the TPC-C. So it genuinely looks more realistic. Second, the read vs. write ratio is much closer to the norm where most activity is heavy reads followed by some writes. Third, the data is based off year 2000 census data rather than simply being randomly generated values within a defined range. Fourth and final, the database design enforces referential integrity that most of today's OLTP applications do by default (in fact, very few relational database professionals would build an OLTP database where there we no foreign keys being enforced). So while the TPC-E might be harder to digest, it's worth the effort since it is more lifelike than the old TPC-C benchmark. However the TPC-C remains the most quoted.

Another aspect of the database design is what advanced physical database options or schema object definition modifications are permissible? The TPC-E spec is fairly clear about what can and cannot be done. The following are allowed:

- Table data row clustering
- Unique indexes for primary key constraints
- Non-unique indexes for foreign key constraints
- Horizontal table partitioning
- Vertical table partitioning
- Replication of tables

Note that the TPC-E allows for two items that the TPC-C did not: table data row clustering and foreign key indexes. Once again this contributes to the general feeling that the database application is more realistic. So if and when you're looking for a realistic OLTP type database benchmark, the TPC-E is a very good choice. While no benchmark is perfect or 100% accurately mimics your OLTP applications, the TPC-E provides a very reasonable facsimile.

TPC-H

The TPC-H is one of my favorite database benchmarks. In fact, I have likely run more TPC-H benchmarks than any other. Many other people routinely use and quote this database benchmark as well. While not quite as popular as the TPC-C, it is probably the second most quoted database benchmark. For the past decade or so while the topics of data warehouses and business intelligence (BI) were hot, the TPC-H flourished. And now with data mining and data analytics, the TPC-H still draws lots of attention.

As usual let's begin by examining at the TPC-H database benchmark definition from the tpc.org web site and the TPC-H spec:

- Website: TPC-H is a decision support benchmark that consists of a suite of business-oriented ad-hoc queries. It illustrates decision support systems that examine large volumes of data, executes queries with a high degree of complexity, and answers critical business questions.
- TPC-E spec: TPC-H is comprised of a set of business queries (that have been given a realistic context) designed to exercise database system functionalities in a manner representative of complex business analysis applications.

For once both the website and spec definitions provide sufficient information to get the basic gist of the benchmark. However here is a more thorough and therefore more useful definition:

> *The TPC-H is a collection of 22 very complex SQL queries typical for a reporting database that supports of an OLTP system. These queries are meant to mimic ad-hoc data analysis queries submitted by end users via business intelligence tools in order to answer critical business questions.*

Note what it does not say: the TPC-H is not a star schema designed data warehouse (that's the TPC-DS, which is covered next). As you examine the TPC-H data model shown in Figure 2-4 you'll see that it's really another simplistic database design much like the TPC-C.

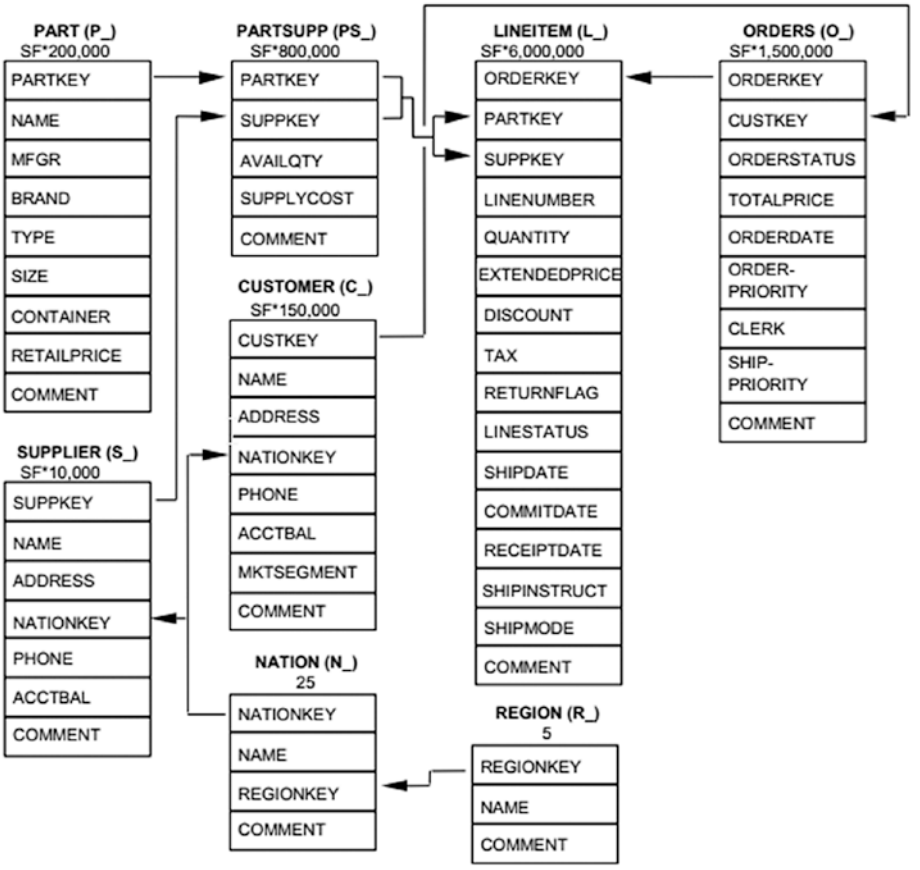


Figure 2-4. *TPC-H Data Model*

If you look closer at Figure 2-4 you'll see that each table has a number of rows dependent on the scale factor. Accordingly the LINEITEM table has 6 million rows per scale factor. So what scale factor should you choose when you run the TPC-H benchmark? The spec does not really say nor should it really as it's intended as a description of the benchmark design requirements. However, the general consensus among thus who routinely run the TPC-H benchmark, you should choose as follows: small = 300; medium = 3,000; large = 30,000; and huge = 300,000. While not 100% true, you can guesstimate about 1GB per scale factor. Therefore. the

approximate database size estimate you could count on is roughly: small = 300GB; medium = 3TB; large = 300TB; and huge = 3PB. Remember that the TPC-H by definition executes 22 complex queries that scan large volumes of data, and as such this database benchmark is highly dependent on the IO subsystem performance. If you wanted to perform a medium sized or larger TPC-H on traditional spinning magnetic disk, then you should have well over 100 spindles. The TPC-H genuinely requires the data being scanned to be housed on and spread across many disks. Of course, new IO technologies such as solid state disk (SSD), all flash disk arrays, PCIe flash disk, and non-volatile memory express (NVMe) offer higher IO capabilities with lower latencies such that there is no golden rule (as of yet) on how to best spread large amounts of TPC-H data across them.

Another aspect of the database design is what advanced physical database options or schema object definition modifications are permissible? The TPC-E spec is fairly clear about what can and cannot be done. The following are allowed:

- Table data row clustering
- Unique indexes for primary key constraints
- Non-unique indexes for foreign key constraints
- Horizontal table partitioning (including multi-level)
- Auxiliary data structures (with some limitations)

Note that the TPC-H only allows horizontal partitioning, and not vertical partitioning. Furthermore, the TPC-H also does not allow replication. However, it's the allowance for auxiliary data structures that is most unique and powerful. Thus common DBA techniques for query optimization such as creating an index that wholly fulfills a query are permitted. Moreover, advanced techniques such as materialized views (i.e., automated aggregate tables) are also allowed and can result in major performance improvements. Note, however, that none of the current

database benchmarking tools offers to leverage such features either automatically or via options. So it's up to you as the DBA to manually add these to the mix. This extra effort is often more than justified.

Finally, to better understand the TPC-H more fully, you really need to review the spec and examine all 22 complex queries. They are all purposefully designed to stress various aspects of the database's internal SQL optimizer and execution engine. Listing 2-1 shows one of the least complex and easy to read TPC-H's queries, the Potential Part Promotion Query (Q20). As per the TPC-H spec this query identifies suppliers in a particular nation having selected parts that may be candidates for a promotional offer.

Listing 2-1. TPC-H Query #20

```
SELECT   s_name, s_address
  FROM h_supplier, h_nation
   WHERE s_suppkey IN (
      SELECT ps_suppkey
        FROM h_partsupp
       WHERE ps_partkey IN (SELECT p_partkey
                  FROM h_part
                   WHERE p_name LIKE 'dark%')
         AND ps_availqty >
            (SELECT 0.5 * SUM (l_quantity)
             FROM h_lineitem
            WHERE l_partkey = ps_partkey
              AND l_suppkey = ps_suppkey
              AND l_shipdate >=
                      TO_DATE ('1997-01-01', 'YYYY-MM-DD')
              AND l_shipdate <
                 ADD_MONTHS (TO_DATE ('1997-01-01',
                            'YYYY-MM-DD'
                           ),
```

```
                        12
                       )))
   AND s_nationkey = n_nationkey
   AND n_name = 'MOROCCO'
ORDER BY s_name
```

The complex nature of these 22 TPC-H queries has yielded one very interesting side effect in terms of intended usage. I have found that these 22 queries are great for comparing the relative database performance increase or decrease from a major release or a minor patch. I also find them just as useful to compare one database vendor's SQL optimizer and engine to another. That's the real reason why the TPC-H is one of my favorites that I use all the time.

TPC-DS

The newest somewhat popular database benchmark is the TPC-DS, which has only been around a little over two years (August 2015). Much like the TPC-E, the TPC-DS tends to be favored by hardware and database vendors. The one notable exception is recent the book *Oracle Database 12c Release 2 Testing Tools and Techniques for Performance and Scalability* by Jim Czuprynski, Deiby Gomez, and Bert Scalzo (Mc-Graw Hill Education, 2017). In that book the authors extensively use the TPC-DS benchmark to showcase valuable techniques for performance optimization and database scalability. In fact the book is literally packed with example queries and workloads from the TPC-DS with recommendations for modifications plus the before and after results. That's because much like the TPC-H the TPC-DS cab be used to compare the relative database performance increase or decrease from a major release or a minor patch plus to compare one database to another.

Once again let's start by looking at the TPC-DS database benchmark definition from the tpc.org web site and the TPC-DS spec:

- Website: TPC-DS is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance. It provides a representative evaluation of performance as a general purpose decision support system.
- TPC-DS spec: the same as the website definition.

As before neither definition really provides any kind of meaningful insight into the nature of the application workload other than saying it models aspects of a decision support system. Here is a more thorough and therefore more useful definition:

> *The TPC-DS schema is designed specifically to simulate a complex data warehouse application workload for a retail product supplier that is providing its customers the ability to generate merchandise orders through its three main sales channels: mailorder catalog, brick-and-mortar stores, and web site. The schema comprises seven fact tables and seventeen dimensions that include handling returns from any one of the three sales channels as well as an inventory tracking function.*

The TPC-E design is very complex, plus it is a true star schema design data warehouse. The seven major stars are shown in Figures 2-5 through 2-11.

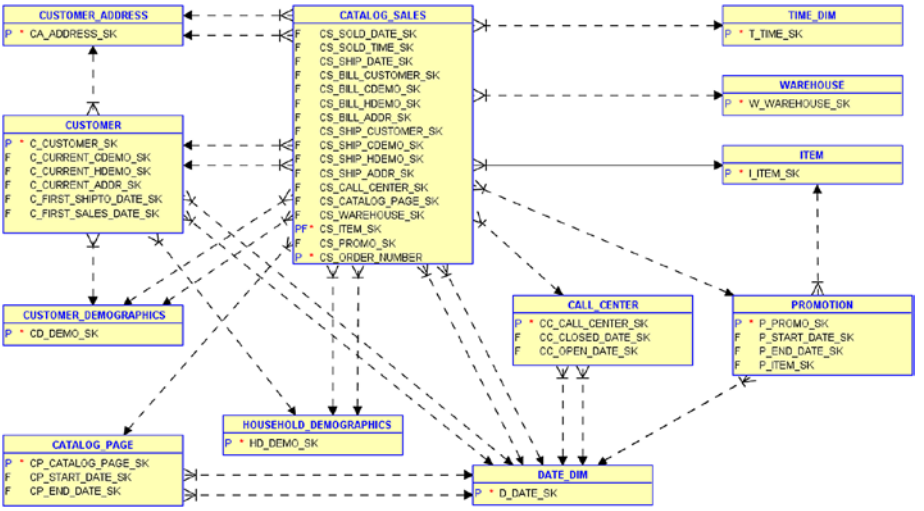


Figure 2-5. *Star for CATALOG_SALES*

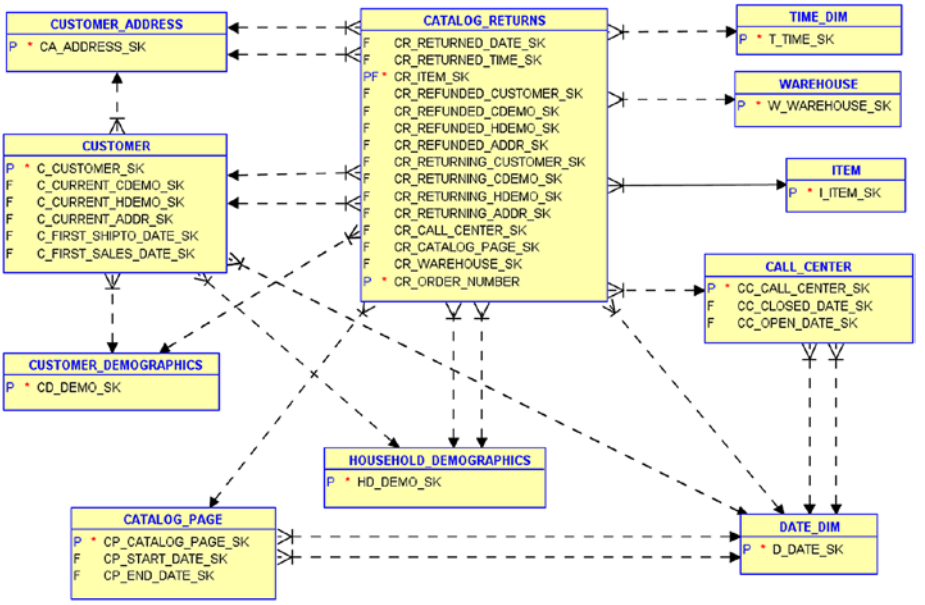


Figure 2-6. *Star for CATALOG_RETURNS*

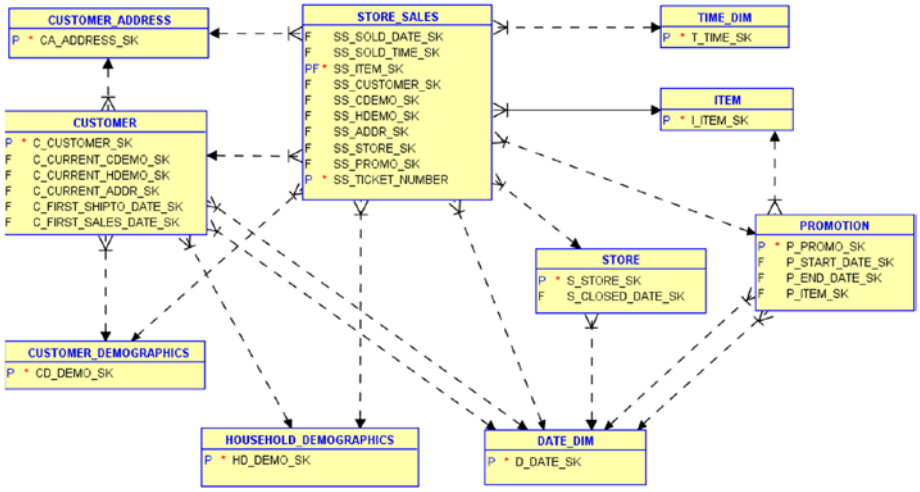


Figure 2-7. *Star for STORE_SALES*

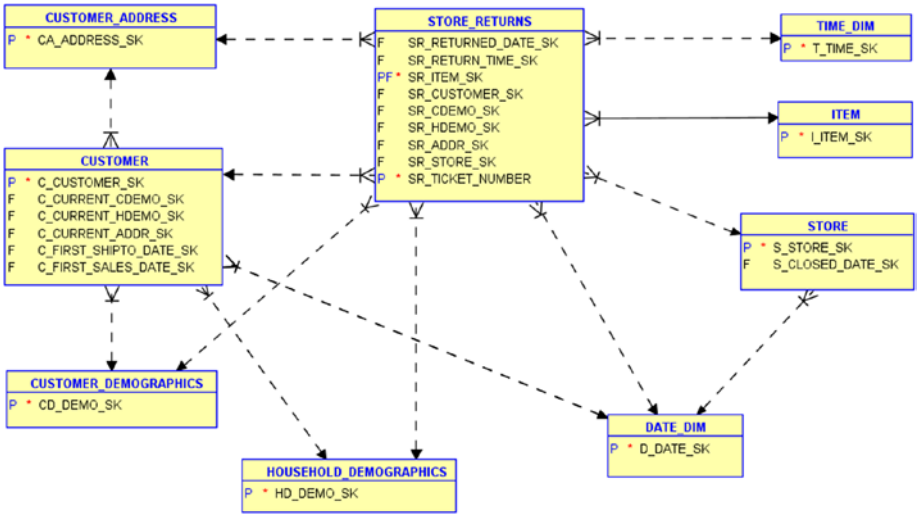


Figure 2-8. *Star for STORE_RETURNS*

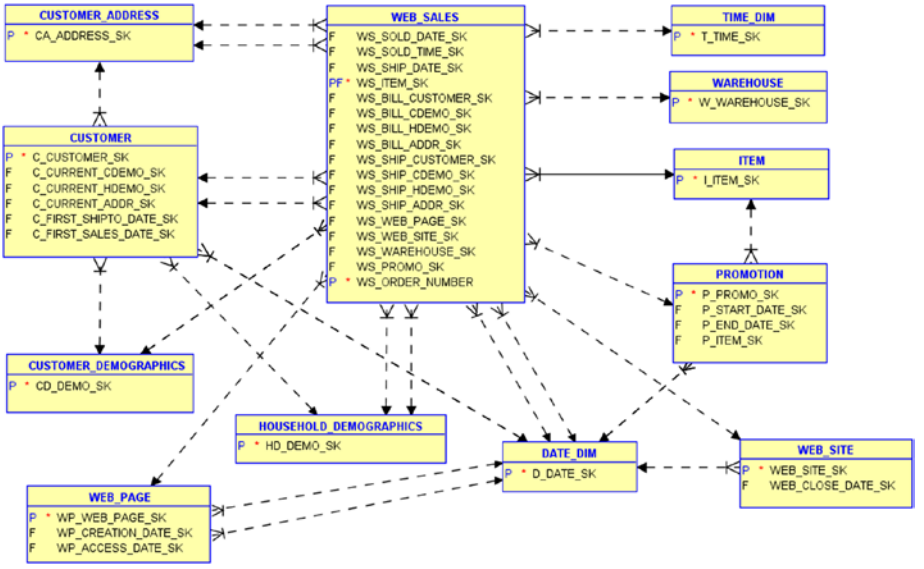


Figure 2-9. *Star for WEB_SALES*

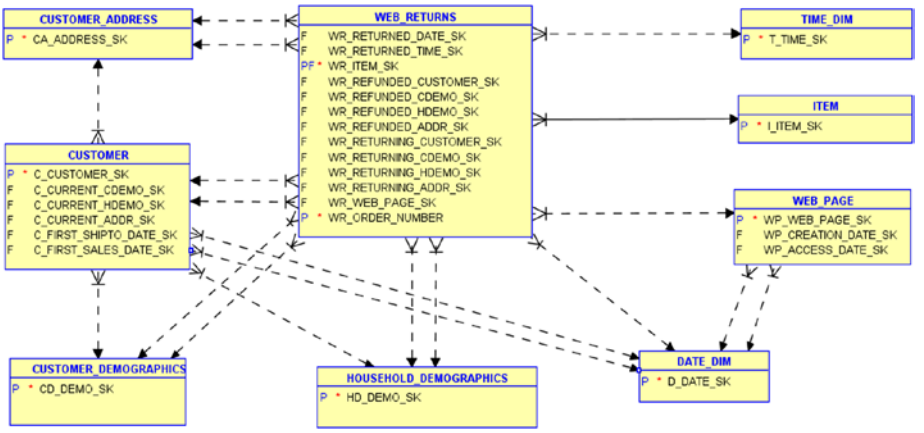


Figure 2-10. *Star for WEB_RETURNS*

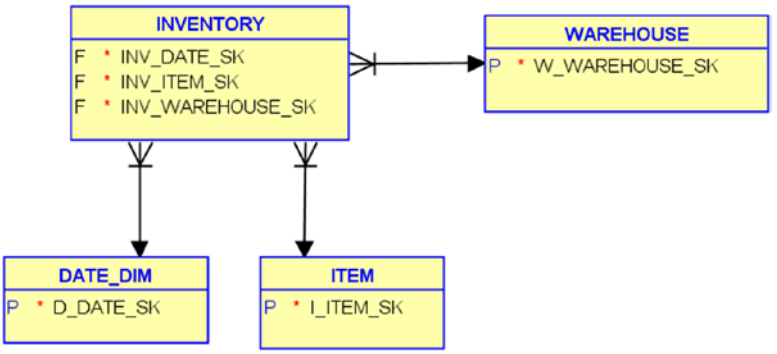


Figure 2-11. *Star for INVENTORY*

Another aspect of the database design is what advanced physical database options or schema object definition modifications are permissible? The TPC-DS spec is fairly clear about what can and cannot be done. The following are allowed:

- Table data row clustering
- Unique indexes for primary key constraints
- Non-unique indexes for foreign key constraints
- Horizontal table partitioning (including multi-level)
- Vertical table partitioning
- Auxiliary data structures (with some limitations)
  - Explicit – auto created as a consequence of a directive
  - Implicit – not created as a consequence of a directive

Note that the TPC-H only allows horizontal partitioning, and not vertical partitioning. Furthermore, the TPC-H also does not allow replication. However, it's the allowance for auxiliary data structures that is most unique and powerful. Thus common DBA techniques for query optimization such as creating an index that wholly fulfills a query are

permitted. Moreover, advanced techniques such as materialized views (i.e., automated aggregate tables) are also allowed and can result in major performance improvements. Note however that none of the current database benchmarking tools offers to leverage such features either automatically or via options. So it's up to you as the DBA to manually add these to the mix. This extra effort is often more than justified.

Finally to better understand the TPC-DS more fully, you really need to review the spec and examine all 100 complex queries. The TPC-DS specification is focused mainly on obtaining business answers from a potentially massive dataset. From that viewpoint, the most important part of the TPCDS benchmark are the nearly 100 individual queries that attempt to provide answers for these particular business questions. Each of the TPC-DS queries implements varying degrees of analytic complexity; some queries access 10 or more fact and dimension tables, thus producing varying and complex execution plans depending upon the values supplied for the query selection criteria. Listing 2-2 shows one of the TPC-DS's queries, the business query 6 (Q6). As per the TPC-DS spec, this query lists all the states with at least 10 customers who during a given month bought items with the price tag at least 20% higher than the average price of items in the same category.

Listing 2-2. TPC-DS Query #6

```
SELECT a.ca_state state, COUNT(*) cnt
FROM  tpcds.customer_address a ,
      tpcds.customer c ,
      tpcds.store_sales s ,
      tpcds.date_dim d ,
      tpcds.item i
WHERE a.ca_address_sk   = c.c_current_addr_sk
  AND c.c_customer_sk   = s.ss_customer_sk
  AND s.ss_sold_date_sk = d.d_date_sk
```

```
  AND s.ss_item_sk      = i.i_item_sk
  AND d.d_month_seq = (SELECT DISTINCT (d_month_seq)
                       FROM  tpcds.date_dim
                       WHERE d_year = 2001
                         AND d_moy    = 6)
  AND i.i_current_price > 1.2 * (SELECT AVG(j.i_current_price)
                           FROM  tpcds.item j
                           WHERE j.i_category = i.i_category)
GROUP BY a.ca_state
HAVING COUNT(*) >= 10
ORDER BY cnt
```

Note that there is currently only one database benchmarking tool that has implemented the TPC-DS benchmark. But given the current market hype on all things related to data analytics, I would expect other tools to follow suit in the near future. However, this benchmark with 7 star schemas and 100 complex queries presents a formidable challenge to implement.

Note The following sections will cover the remaining less popular or obsolete database benchmarks in lesser detail. That doesn't necessarily mean they are not as complex or not worthwhile, just that the prior database benchmarks are more common and thus ones that readers are likely to perform.

Less Popular Benchmarks

TPC-DI

Looking back over the popular database benchmarks we have tests for OLTP, general reporting, and star schema data warehouses. What's missing is loading data via batch jobs into reporting systems and data warehouses? In fact, over the years the sheer amount of data being loaded and retained has mushroomed. So much so that term extract, transform, and load (ETL) has been replaced with the more comprehensive term data integration (DI). And so the TPC-DI benchmark was born to address this special and important niche. The conceptual model of the TPC-DI system architecture is shown in Figure 2-12.

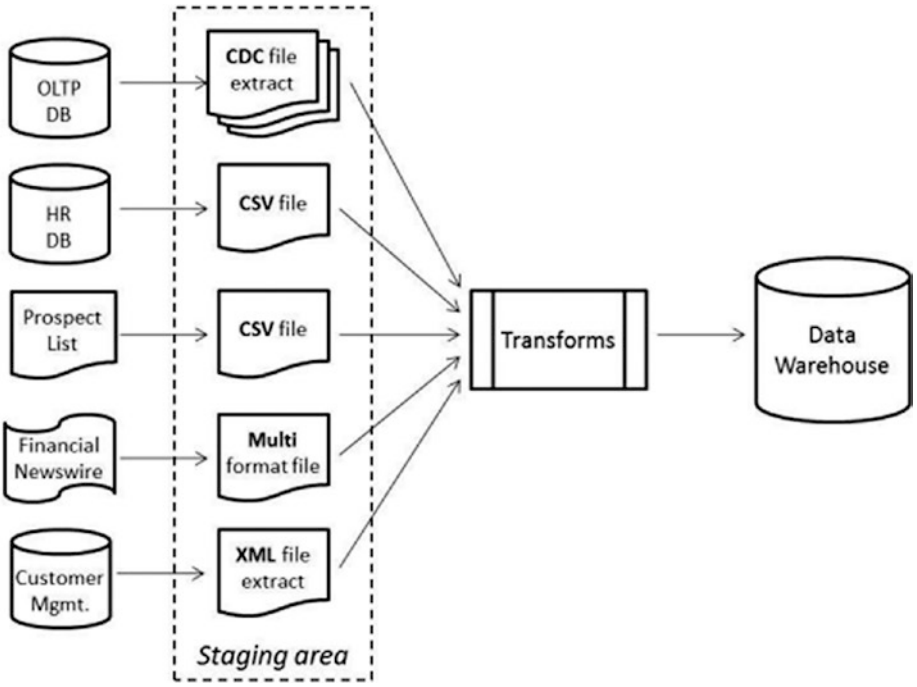


Figure 2-12. *TPC-DS System Conceptual Model*

This diagram should not surprise anyone as it's the typical architecture diagram shown whenever creating a data warehouse. Some people might skip the staging area concept and load directly into the data warehouse, hence why the staging area is surrounded by a dotted line (i.e., optional). The target data warehouse design is a traditional dimensional model, commonly referred to as a star schema. This approach was best described in 2002 by Ralph Kimball's book *The Data Warehouse Toolkit* (Wiley). However, Ralph actually first proposed such design while he was at Redbrick, an early entrant into the data warehouse database vendor space. I actually met Ralph back around 1995 when he was doing live seminars. He taught me star schema and I taught him data modeling (I was working for Logic Works, the maker of the Erwin data modeling tool). I also wrote a book in 2003 based on what I learned in 1995 and then practiced for over eight years titled *Oracle DBA Guide to Data Warehousing and Star Schemas* (Prentice Hall). While that book may be 15 years old now, the techniques espoused remain relevant even today.

The TPC-DI data warehouse's basic conceptual data model is shown in Figure 2-13.

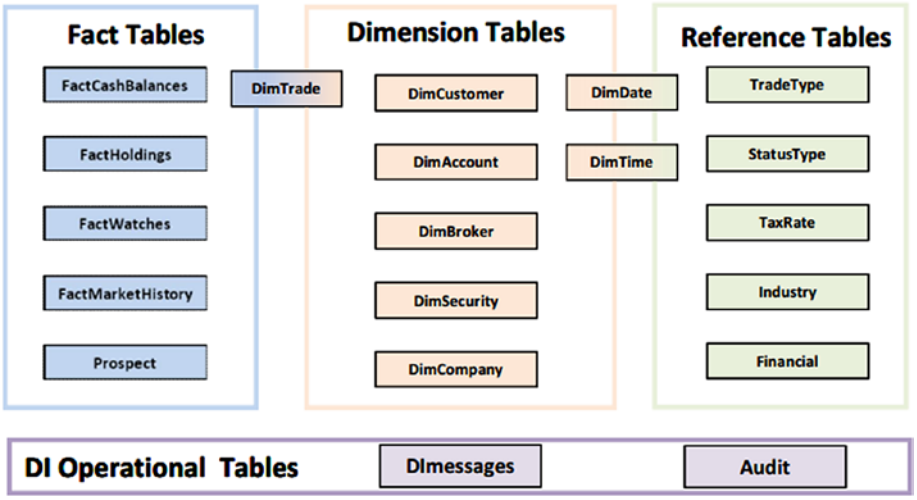


Figure 2-13. *TPC-DS Conceptual Data Model*

For those new to data warehousing, two different terms show up. Dimension tables are smaller, de-normalized tables containing business descriptive columns that end users query on. It's not uncommon for these tables to be fully indexed to best support ad hoc queries. Fact tables are very large tables with primary keys formed from the concatenation of all related dimension table foreign key columns, and possessing numerically additive, non-key columns used for calculations during end-user queries.

TPC-VMS

With the virtualization craze the past decade, it was inevitable that databases, even large, mission critical ones, would end up running as a virtual machine (VM) on a Hypervisor (e.g., VMware). SQL Server database administrators and those on open source databases readily embraced this move. On some database platforms such as Oracle the database administrators resisted at first and then were slow to fully adopt. However, with time and now databases in the cloud, few if any database administrators have any reservations about placing their database on a VM. That resulted in the need for a reliable database benchmark for a virtual machine environment. Remember that a Hypervisor on a server may have many VM's running concurrently, thus special considerations for database consolidation of dissimilar types of databases must also be handled.

Introduced in 2012 the TPC-VMS was designed to meet this special need. The TPC-VM benchmark leverages the popular benchmarks from the prior section (i.e., TPC-C, TPC-E, TPC-H, and TPC-DS). However, the TPC-VM considers its version of each test to be unique, and thus comparable neither to each other nor to the base benchmarks from which they were derived. When you instantiate a TPC-VM benchmark you select one of the four, and then three VM's each running the same size database for the chosen benchmark are run. The way the results are calculated is also different. But since it'd not very popular we'll forgo those. They are defined in full in the spec.

The only shortcoming in my opinion is that I would rather have had the TPC-VMS run three different benchmarks of the four or even all four so that dissimilar workloads could be tested. Maybe this will be addressed in a future spec update or possibly even a whole new benchmark (since the scoring would probably have to be different).

Wisconsin

The Wisconsin benchmark is one of the oldest industry standard database benchmarks. It came into existence circa 1981, around the time that new SQL-based relational databases were coming onto the scene. I include the Wisconsin benchmark in this book because of its significant historical relevance. It came from a period when computers were anemic, disk space costs were high, and relational databases were just emerging. As such the Wisconsin's schema has just three tables: one with just 1,000 rows and two with 10,000 rows each. The total database size was just 5 megabytes. Moreover it contained a total of 32 SQL commands (26 inserts, 2 updates, 2 deletes, and 2 selects). In those early days with new SQL database technology and limited computer resources this test was probably quite reasonable. But by today's computing standards it's simply far too anemic of a test to use for anything meaningful. However as one of the very earliest database benchmarks and a precursor to those that followed, it's at least worth knowing about.

AS^{3}AP

The ANSI SQL Standard Scalable and Portable (AS^{3}AP) benchmark is another fairly old industry standard database benchmark. It came into existence circa 1984 and was in many respects an extension of the Wisconsin benchmark. I include it in this book because you can still find references to it and one of the database benchmarking tools still offers it. The AS^{3}AP's schema has just five tables: one with 1 column and 1 row, and

four with from 10,000 to one billion rows each based on the scale factor. As such its total database size was from 4 megabytes to 400 gigabytes (once again very small by today's standards). You would choose the maximum scale factor that would complete the benchmark in 12 hours or less (with a goal being to run as close to 12 hours as possible). The benchmark consisted of both single and multi-user workloads, plus included all the time to create, load, and index the database. The only measurement required by this benchmark is total query elapsed time, within the 12-hour window limit.

Obsolete Benchmarks

TPC-A

The TPC-A benchmark was the first official benchmark that spawned from the tpc.org effort. It was a first-generation OLTP database benchmark that strived to approximate a typical bank's transaction workload with an emphasis on updates (since the workload is skewed and kept artificially simple as well, the TBC-A does not represent a true OLTP benchmark). The TPC-A had just four tables: ACCOUNT, BRANCH, TELLER, and HISTORY as shown in Figure 2-14. For each branch you would have 10 tellers, 10 terminals, and 100,000 accounts. It originated in 1989 and was deemed obsolete by 1995.

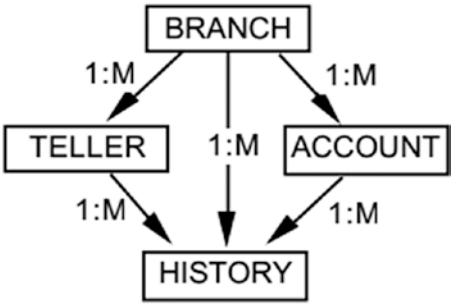


Figure 2-14. *TPC-A Data Model*

TPC-B

The TPC-B benchmark is actually a database stress test (the difference in these terms was covered in Chapter 1). It was specifically designed to be a stress test on the core portion of a database system (such as IO bandwidth and processing time). It is essentially the TPC-A having the same simple tables shown in Figure 2-14 but with a radically different and artificial transaction workload. The TPC-B uses a single, simple, update-intensive transaction to stress the database. It stresses the memory and disk I/O subsystems by causing large amounts of small read and write jobs. It originated in 1990 and was deemed obsolete by 1995.

TPC-D

The TPC-D benchmark is the first intended for data warehouse type workloads. In many respects it was a precursor to the TPC-H benchmark (even having the same database design – refer back to Figure 2-4). The primary difference was that it had just 17 queries instead of 22. It originated in 1989 and was deemed obsolete by 1999.

TPC-R

The TPC-R benchmark is an unrestricted version of the TPC-H benchmark (same 22 queries and same database design – refer back to Figure 2-4). The primary difference is it allowed for unrestricted indexing and partitioning in order to allow people to more fully leverage database features. It originated in 1999 and was deemed obsolete by 2005.

TPC-W

The TPC-W benchmark is arguably not even a real database benchmark, but rather a web application benchmark that includes a database. In fact the spec even allows replacing the database with any data store you like, even including a file system solution. The workload nature is that of an internet commerce environment that simulates activities of a business-oriented transactional web server. It originated in 2003 and was deemed obsolete by 2005. Shown in Figure 2-15, it is included here only for completeness.

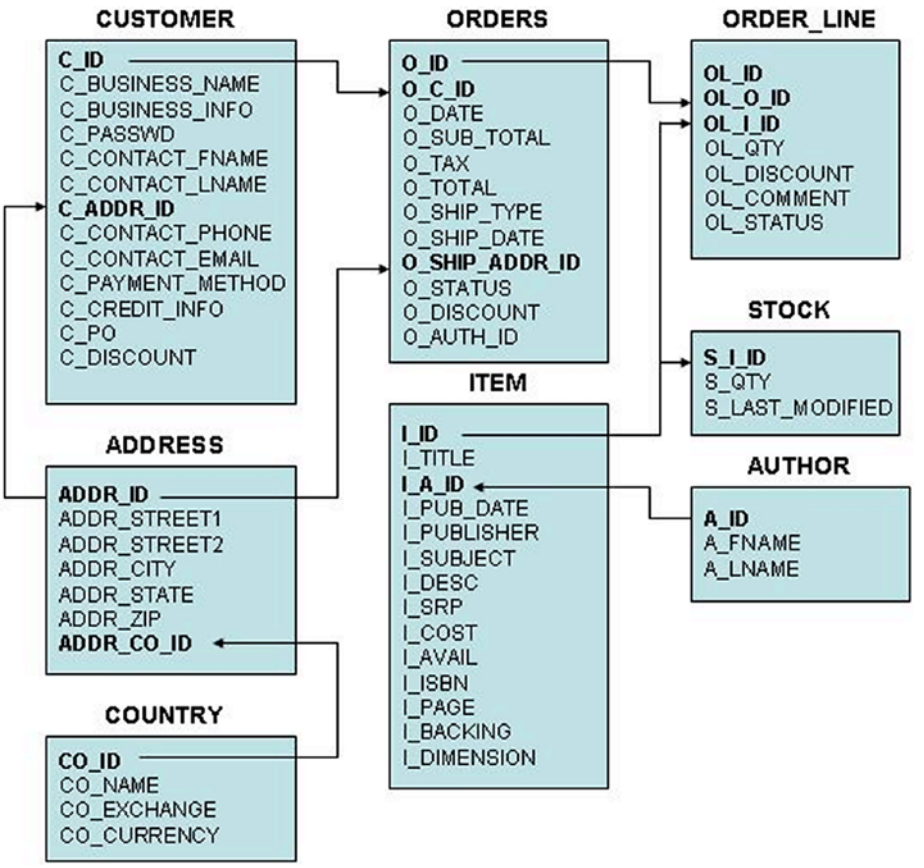


Figure 2-15. *TPC-W Data Model*

Summary

So many benchmarks, so little time! In this chapter we reviewed all the various database benchmarks, their origin, and their general purpose. The reason for spending an entire chapter on this history and context is to equip the reader to intelligently select from among the many benchmarks those that most align with their interests. Moreover, just enough information is provided to encourage the reader to seek out and read the chosen database benchmark's specification document where far more

detailed information is provided. For example, the TPC specs all include an appendix that details the SQL DDL to create all the database objects. So while this book can point the reader in a proper direction, only the actual spec can answer all questions. Therefore, they are must-read materials for success.

CHAPTER 3

Benchmarking Tools

In Chapter 1 we differentiated between a database benchmark, database stress test, and workload capture/replay. Then in Chapter 2, we covered all the relevant database benchmarks, both current and past (i.e., obsolete). So much like a teen learning to drive a car, you've now studied the rules of the road and passed the written test to obtain a learner's permit. Now it's time to exercise that knowledge and to do some actual driving (long before ever taking the final road test). For that, the anxious and excited teen would need the use of a car and a legal driver as a passenger (note that there are other limitations such as no other teens and no driving at night). So similarly for database benchmarking, you are going to need some tools and to do some test driving with limits until you get sufficient experience before doing it for real. Thus, here in Chapter 3, we'll cover some of the database benchmarking tools that are available and how to best use them. Furthermore, we'll cover some other tools that you may consider a benchmarking tool but which better fit under the category of database stress test or workload capture/replay tools.

The car driving example also applies as it relates to how complex are the various database benchmarking tools to actually use. Some cars have an automatic transmission while others have a manual. Most people would agree that the automatic transmission is easier to drive, but that a manual one with a sophisticated driver should be able to get better gas mileage. Likewise, we all pretty much prefer a car with both power-assisted steering and brakes, as again such features make driving much easier.

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_3

The same is true for current database benchmarking tools; some are far more automated and thus easy to use while others are more difficult but might offer greater flexibility. Therefore, proper preparation for the type of tool to be used is required. Moreover, as with a car, you should be able to utilize either type to get to your destination. Plus of course, your mileage and experience may vary.

Finally, note that while this chapter aims to provide you with all mandatory basics plus some useful tips and tricks, it quite logically takes both time and perseverance to become an expert who can reasonably test advanced scenarios to obtain reliable and repeatable results. Note that "*repeatable*" is the hardest and more important goal to achieve. Just because a car can go 125 MPH one time does not mean it can readily repeat that feat. Many variables come into play during such an effort. The results must correlate to all the variables, plus the conditions must be repeatable within some reasonably close range. Anything else is just a very lucky "*one-off*" result of little to no value.

Storage Benchmarking

Before looking at database benchmarking tools, we must first recognize those tools that are not classified as true database benchmarking tools even though they are quite popular. Sometimes database administrators, often working with storage administrators, will concentrate on IO rates per second or IOPS. They are more interested in how fast the disk drives can read and write data than the database engine itself. So they are not interested in actual database transactions, but rather raw IO rates. As an example, you cannot really know just how fast a car can travel by knowing the engine RPM rate. Likewise you cannot know the actual database performance characteristics by knowing just the disk IOPS. Yes, the faster the IOPS of the disks the faster the database might be, but you really can only guesstimate the database performance based upon IOPS alone.

While a contributing factor, it's not the only one, nor even the most significant in many cases. Just as with the automobile, we need to know other factors than simply the RPM's, such as the transmission gearing and tire size. Nonetheless these tools remain highly popular even if they can only provide the most basic idea of true database performance.

WINSAT

If your database is on a Microsoft Windows box, then a very simple built-in IO testing tool is the WINSAT command. On my Windows box my database data files are located on the D: drive, which is a Samsung SATA-III SSD; therefore the command would be **`winsat disk -drive D`** (note: do not put a colon after the drive letter). Figure 3-1 shows an example of the output.

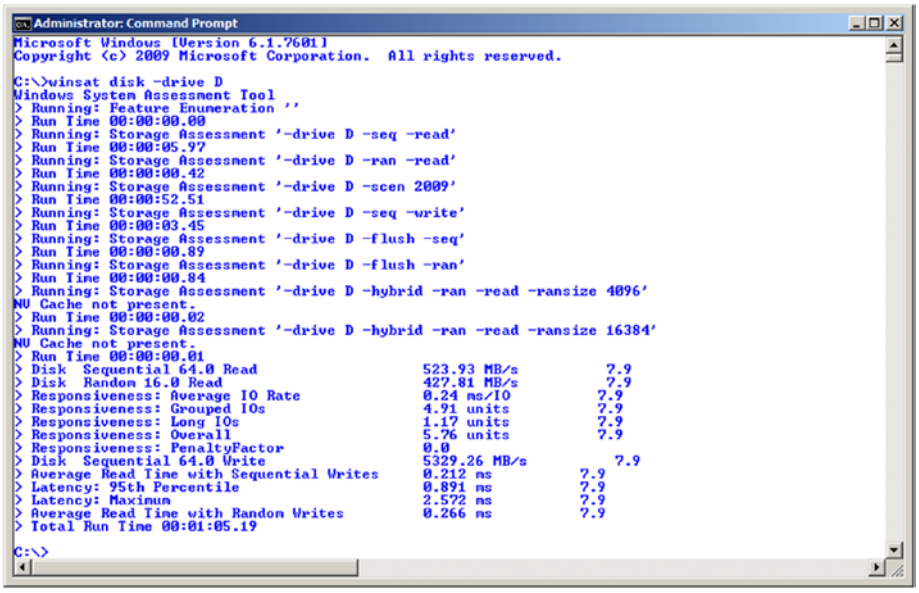


Figure 3-1. *Windows WINSAT Utility*

HDPARM

If your database is on a Linux box, then a very simple built-in IO testing tool is the hdparm command. On my Linux box my database data files are located on the /disk2 drive, which is a Samsung traditional SATA-III disk; therefore the command would be **`hdparm -x /dev/sdb1`**. Figure 3-2 shows an example of the output.

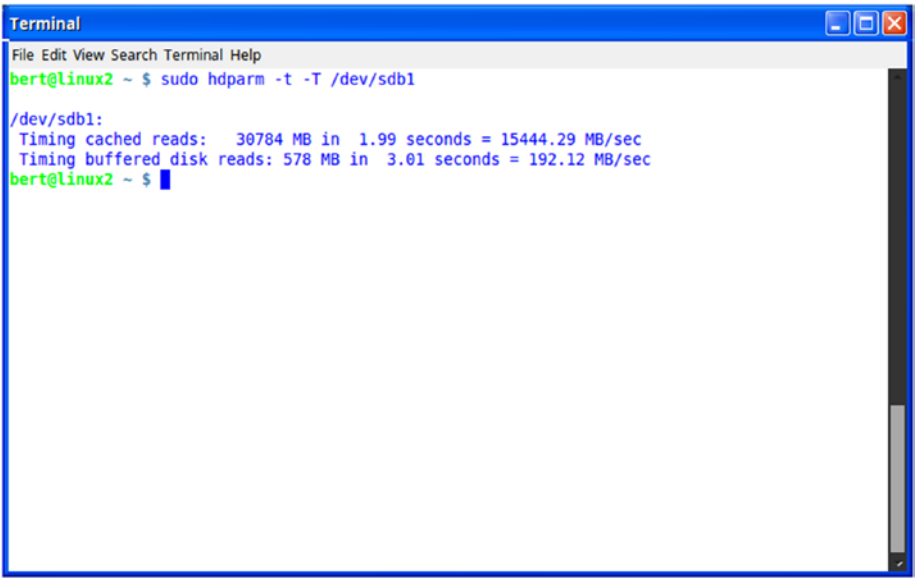


Figure 3-2. *Linux HDPARM Utility*

SQLIO

If your database is on a Microsoft Windows box, then a deprecated but still popular IO testing tool is the SQLIO utility. While it's no longer available on the Microsoft website, many freeware sites still list it. On my Windows box my database data files are located on the D: drive, which is a Samsung SATA-III SSD; therefore the command would be **`sqlio -dD`** (note: do not put a colon after the drive letter). Figure 3-3 shows an example of the output.

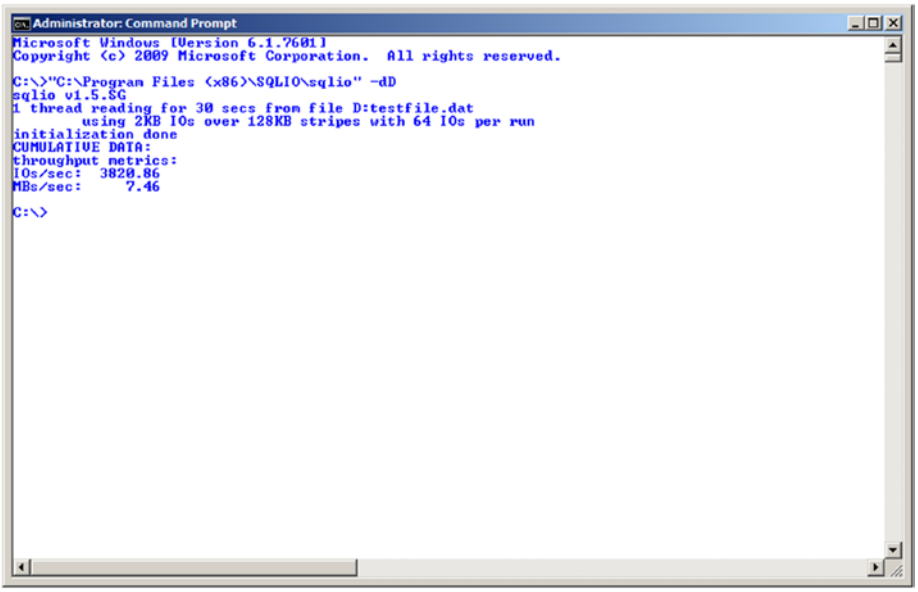


Figure 3-3. *Microsoft SQLIO Utility*

DISKSPD

If your database is on a Microsoft Windows box, then the current and popular IO testing tool is the DISKSPD utility. You can obtain it off the Microsoft website at `https://aka.ms/diskspd`. On my Windows box my database data files are located on the D: drive, which is a Samsung SATA-III SSD; therefore the command would be **`diskspd d:`**. Figure 3-4 shows an example of the output.

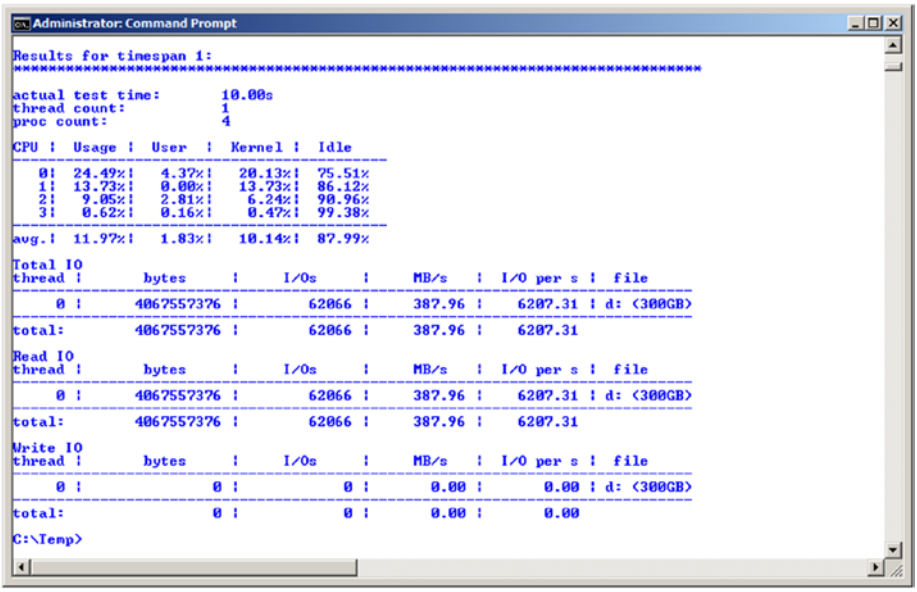


Figure 3-4. *Microsoft DISKSPD Utility*

Database Simulation

The concept of database simulation is fairly simple; the database vendors know what kinds of IO patterns their database engines generally produce, so they can offer tools to simulate those IO patterns for testing without actually creating and benchmarking databases. While such offerings are not true benchmarking tools, they nonetheless offer a valuable insight for some DBAs who want to examine lower-level performance at the IO level – often with an eye toward IOPS.

SQLIOSTRESS

If your database is Microsoft SQL Server, then a deprecated, but still well remembered database simulation tool is the SQLIOSTRESS utility. It's no longer available on the Microsoft website, and I also could no longer locate

it on any freeware site. I only mention it here for completeness as it has been replaced by SQLIOSIM.

SQLIOSIM

If your database is Microsoft SQL Server, then the current database simulation tool is the SQLIOSIM utility. You can do a web search to obtain the Microsoft web site URL from where to download the current version. Note as well that SQLIOSIM does not require SQL Server to even be installed as it totally simulates the IO commands that SQL Server would request from the IO subsystem. SQLIOSIM is a very robust tool, but it has too many options to fully discuss here. For the simplest deployment, just unzip the download file into a directory. Also unzip the configuration file that was also contained in that main zip file. Then copy the default config file **`sqliosim.default.cfg.ini`** to **`sqliosim.cfg.ini`**. Finally edit that copied file and near the bottom uncomment the two FILE sections to specify both a SQL Server data and log file on the device to be simulated against. Now you're ready to go. You can run this utility in command line by running sqliosim.com, and in graphical mode by running sqliosim.exe as show in in Figure 3-5. This utility takes quite a while to run (unless you reduce the TestCycleDuration value in the config file) and the log file contains a detailed HTML report on the results.

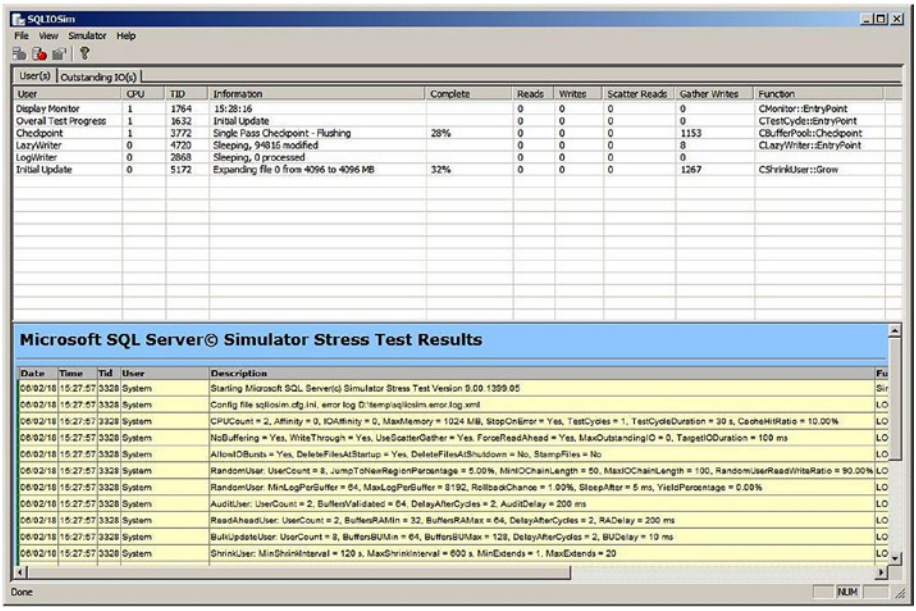


Figure 3-5. *Running SQLIOSIM in Graphical Mode*

Orion

If your database is Oracle, then the one well-known database simulation tool is the ORION utility (where ORION stands for ORacle IO Numbers). It is found in the database and/or grid $ORACLE_HOME/bin directory. And although it's located in an Oracle installation directory, it really does not require the Oracle database to be present. Orion is a very robust and capable tool, so it's advisable to read the Orion manual before doing any detailed testing, but for this book we can look at a simple example. On my Windows box my database data files are located on the D: drive, which is a Samsung SATA-III SSD; thus the command would be **`$ORACLE_HOME\orion -run normal -testname d_drive`**, where the test name parameter specifies the text file with a .lun suffix, which is simply a list of the LUN's

or disk drives to test. On Linux the file contents would be of the form /dev/sdb1, whereas on Windows of the form \\.\D:. Figure 3-6 shows an example of the output. Note the two key values achieved of 514 MBS and 10,472 IOPS.

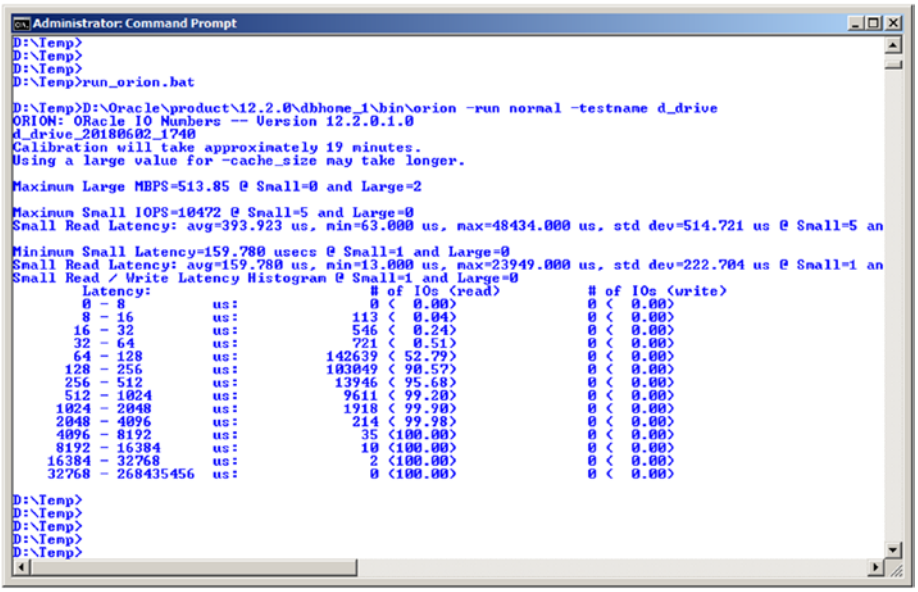


Figure 3-6. *Running Oracle ORION*

Orion will also create a collection of text .TXT and spreadsheet .CSV files, the latter of which you can use to plot the detailed resulting data. The file names will be of the form:

- test_name_date_time_summary.txt
- test_name_date_time _iops.csv
- test_name_date_time _mbps.csv
- test_name_date_time _lat.csv
- test_name_date_time _trace.txt
- test_name_date_time _hist.txt

Note The CSV files output by Orion can very easily be imported into a Microsoft Excel spreadsheet as a graph.

Calibrate IO

If your database is Oracle, then the one very popular database simulation tool is the CALIBRATE_IO utility. According to the Oracle Database Performance Tuning Guide: "This procedure issues an I/O intensive read-only workload (made up of one megabyte of random of I/Os) to the database files to determine the maximum number of IOPS and MBPS that can be sustained by the storage subsystem." The code to call this utility is very simple as shown below.

Calling Oracle's Calibrate IO Utility

```
SET SERVEROUTPUT ON
DECLARE
    lat  INTEGER;
    iops INTEGER;
    mbps INTEGER;
BEGIN
-- DBMS_RESOURCE_MANAGER.CALIBRATE_IO (physical disks,
            max latency, iops, mbps, lat);
    DBMS_RESOURCE_MANAGER.CALIBRATE_IO (1, 10,
            iops, mbps, lat);
    DBMS_OUTPUT.PUT_LINE ('max_iops = ' || iops);
    DBMS_OUTPUT.PUT_LINE ('latency  = ' || lat);
    dbms_output.put_line('max_mbps = ' || mbps);
end;
/
```

On my Windows box my database data files are located on the D: drive, which is a Samsung SATA-III SSD; thus I called CALIBRATE_IO utility with the first parameter set to one disk, and the second parameter set to for a latency maximum of 10 ms. Obviously your database server will likely have far more disk drives and with new technologies such as NVMe flash disks, your latency may also be set far lower. You will therefore have to experiment based on your environment. I ran the CALIBRATE_IO on my notebook and got the results shown in Figure 3-7. Note the reported values of 528 MBS and 54,767 IOPS.

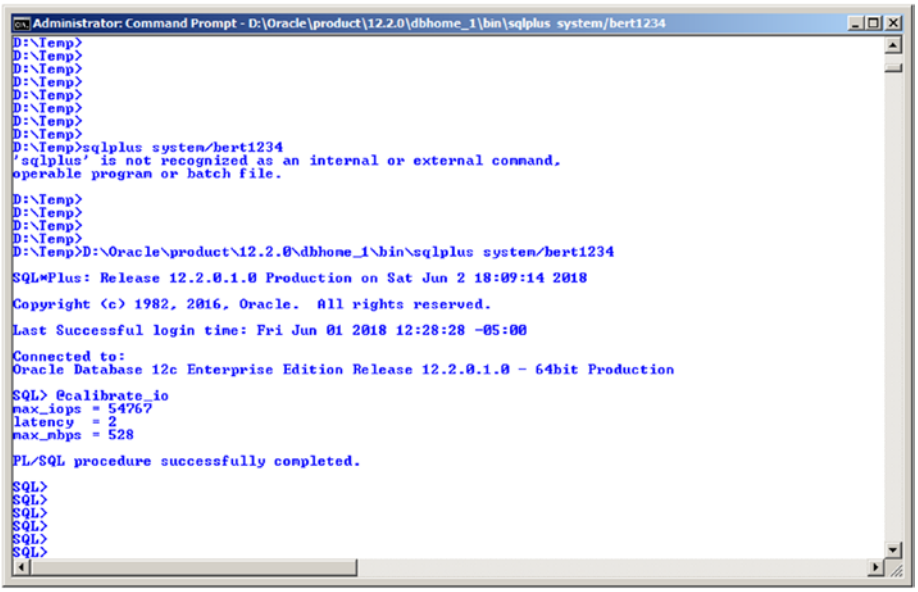


Figure 3-7. *Running Oracle CALIBRATE_IO*

Now I don't want to discourage anyone from using CALIBRATE_IO if it makes sense for your particular situation. But you will find many authors and bloggers who will list numerous shortcomings with this utility's approach.

Here are a few paraphrased examples of what you will find said across the web about this utility (and with which I fully agree):

- CALIBRATE_IO might better be described as a simplified version of Oracle's ORION tool integrated inside the database
- CALIBRATE_IO results of the max IOPS, max MBPS and observed latency values are not always accurate
- CALIBRATE_IO results are neither reliable nor repeatable
- CALIBRATE_IO performs single-block random reads with asynchronous I/O calls buffered in the process heap which has nothing in common with the vast majority of single-block random I/O is known as db file sequential read – which is buffered in the SGA and is synchronous I/O
- Neither CALIBRATE_IO nor Orion access the data that is being read from storage, all they really do is perform the I/O and let the data in the buffer remain untouched

Nonetheless, you will find many DBAs who rely upon and quote this utility solely as if the numbers were somehow gospel. So even though this tool is easy to use with simple output, don't get caught in the trap of overreliance upon a single, limited use case tool such as CALIBRATE_IO. Remember that easy often does not equate to good.

Database Benchmarking

HammerDB

One of the best known and most popular database benchmarking tools that works with multiple database platforms is HammerDB, formerly known as HammerORA from when it was an Oracle-only tool. I have used this tool for many years and believe it to be among the best database benchmarking tools for those wishing to quickly execute and score an industry-standard database benchmark, especially for those who do so only infrequently and thus are not database benchmarking experts. While HammerDB's graphical user interface (GUI) takes a little getting used to, the workflow is simple enough for most to get through without having to read the documentation.

HammerDB is an open source database benchmarking tool for Oracle, SQL Server, DB2, TimesTen, MySQL, MariaDB, PostgreSQL, Greenplum, Postgres Plus Advanced Server, Redis, Amazon Aurora and Redshift, and Trafodion SQL on Hadoop. You will find HammerDB's web page, source forge project, and Git source code version control repository at the following URLs respectively.

- `http://www.hammerdb.com`
- `https://sourceforge.net/projects/hammerora`
- `https://sourceforge.net/p/hammerora/code/ci/master/tree/`

HammerDB offers both Windows and Linux versions for both 32-bit and 64-bit platforms. The current 2.23 version debuted in June of 2017, so it's due for an update as it historically has had one or two a year. When you run the installer on Windows, it will install the software under the **C:\Program Files** or **C:\program Files (x86)** directory as you would expect (based upon the bit size of the version being installed). Then to run the software, you just double-click on the **hammerdb.bat** file. When you

run the installer on Linux it installs in the ~/ or **$HOME** directory. Then to run the software you just double-click on the **hammerdb.tcl** file. When the HammerDB software launches, it displays the screen shown in Figure 3-8.

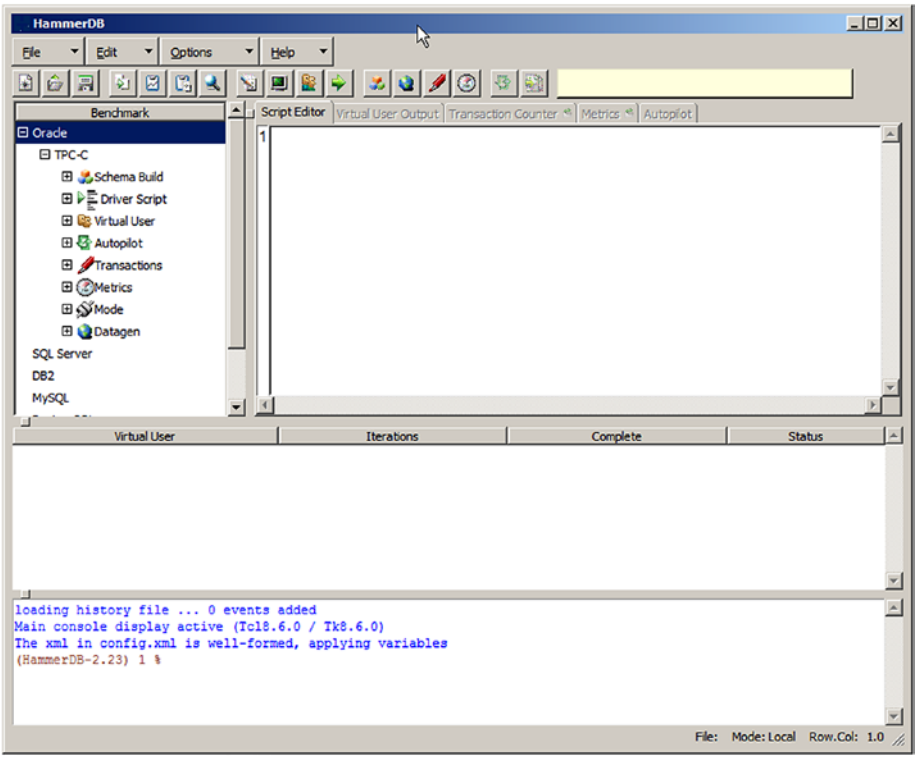


Figure 3-8. *HammerDB Program Launch*

By default, HammerDB assumes that you want to perform a TPC-C benchmark against an Oracle database. These default settings are contained in an XML file named **config.xml** in the HammerDB base install directory. We'll look at modifying this configuration file later, after we get a better feel for the various kinds of options and settings that are available. Let's assume for now that we simply want to instead perform a TPC-H benchmark for an SQL Server database. For that we would simply choose the Main Menu ➤ Options to get the pop-up window shown

in Figure 3-9. Then we'd simply select SQL Server as the database and TPC-H as the benchmark. Note as well that while HammerDB offers many database platforms, it currently offers just the two industry-standard database benchmarks, the TPC-C and TPC-H. However, the website FAQ states that the TPC-E is on their roadmap and planned for a future version.

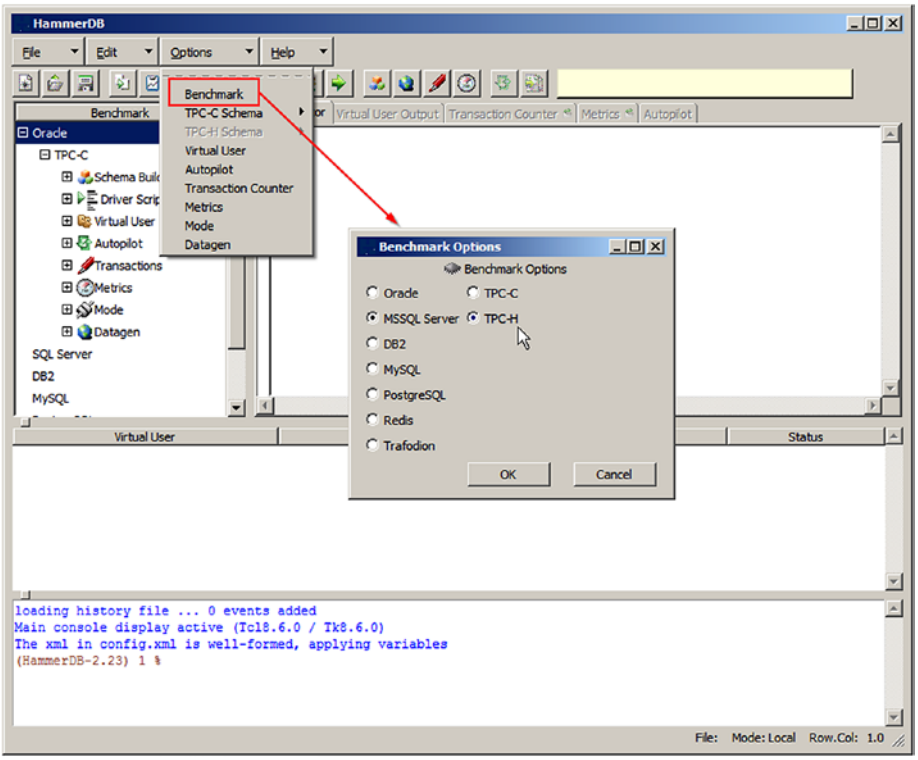


Figure 3-9. *HammerDB Select Database and Benchmark*

Now that we have our database platform and benchmark selected, it's time to dive into the four steps for successfully running a database benchmark test using HammerDB:

1. Build the database schema objects and load the data.
2. Load the driver script whose code implements the selected database benchmark.
3. Create all the concurrent virtual user sessions that will run the selected database benchmark's workload.
4. Initiate all the concurrent virtual user sessions to now run the selected database benchmark's workload.

Step #1: Build Database Objects

So we begin by expanding the "*Schema Build*" tree-view node and then double-clicking on the "*Options*" node, which launches the benchmark schema build options as shown in Figure 3-10.

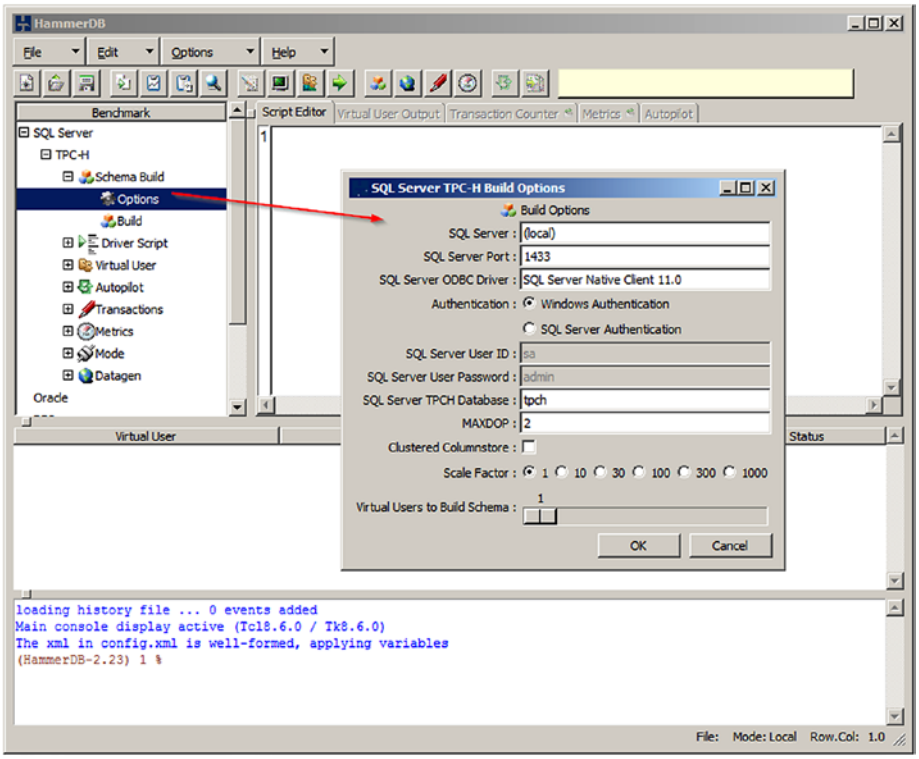


Figure 3-10. *HammerDB Schema Build Options*

Here we simply define the database connection criteria, which database to create the objects in, the maximum degree of parallelism (MAXDOP), and the scale factor for the database benchmark (remember it was explained in Chapter 2 that for the TPC-H, each scale factor equates to about one gigabyte). Had we instead chosen to run the TPC-H database benchmark on Oracle, then the fields presented on this screen would have been slightly different as shown in Figure 3-11. While this screen can vary by database platform, most database administrators will find all of the requested information fairly obvious.

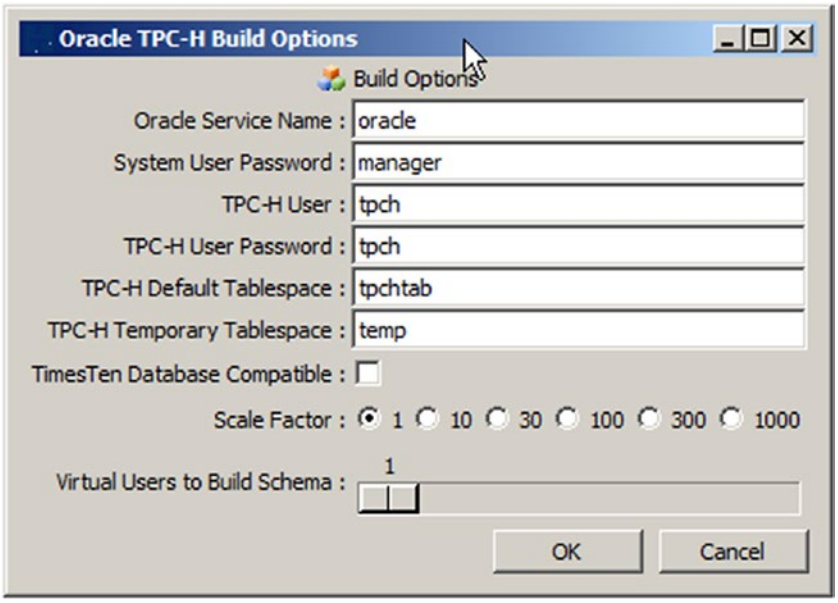


Figure 3-11. *HammerDB Oracle Build Options for Oracle*

With the schema build options now selected, it's time to simply initiate the build process controlled by those selected options. For that we again look under "*Schema Build*" tree-view node, but now double-click on the "*Build*" node, which launches the benchmark schema build (and load) execution as shown in Figure 3-12.

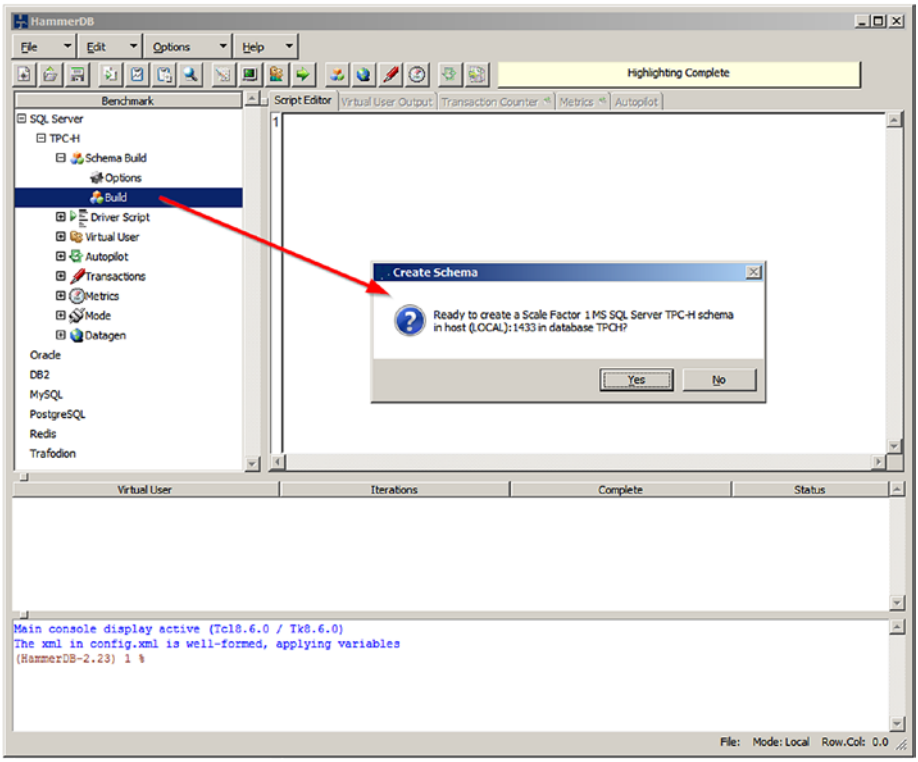


Figure 3-12. *Initiate Building & Loading Database Objects*

Before choosing "Yes" and actually creating and loading the database benchmark objects, it's important to revisit the build options and focus on the two numbers highlighted near the bottom of the screen as shown in Figure 3-13. These are the "*Scale Factor*" and "*Virtual Users to Build Schema*" settings.

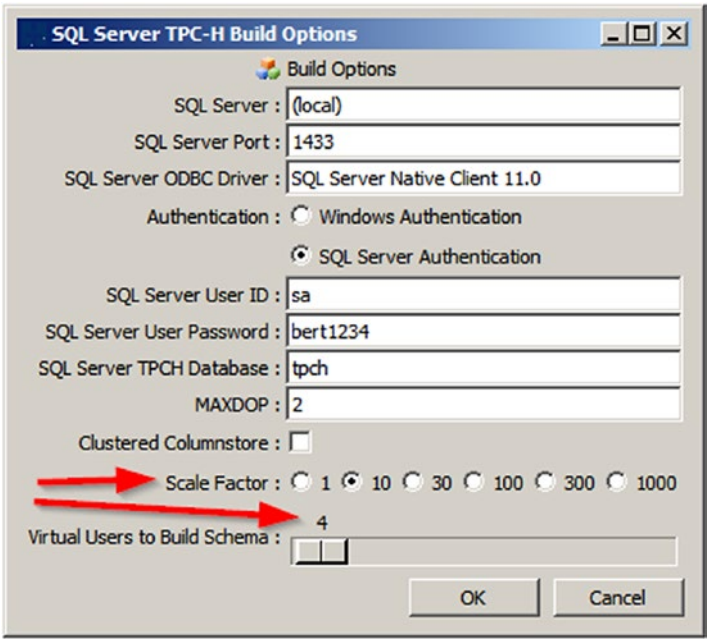


Figure 3-13. *Critical Build Schema Options*

The scale factor is simply the size of the database to create, and for the TPC-H it's about one gigabyte per scale factor. For simple laptop or desktop computer tests for the limited purpose of learning a small-scale factor is OK. However, most database benchmarking experts argue that only scale factors of at least 100 GB's are noteworthy. In fact most published results are for either 300 GB's or 1 TB. As you increase the scale factor you will definitely want to spawn additional concurrent sessions to load that data; therefore you'll want to specify at least four sessions (depending of course on you database server and its IO bandwidth). Please note in Figure 3-13 the scale factor is set to 10 GB and 4 concurrent sessions to load the data. The second number, the number of concurrent sessions to build the schema, will directly affect the build process run time and how it's displayed during execution.

Now that we've revisited these TWO critical value settings, it's OK to initiate the creation process by choosing yes in Figure 3-12. This will result in display of the build logging window as show in Figure 3-14. This window is a little complicated and thus deserves a detailed review.

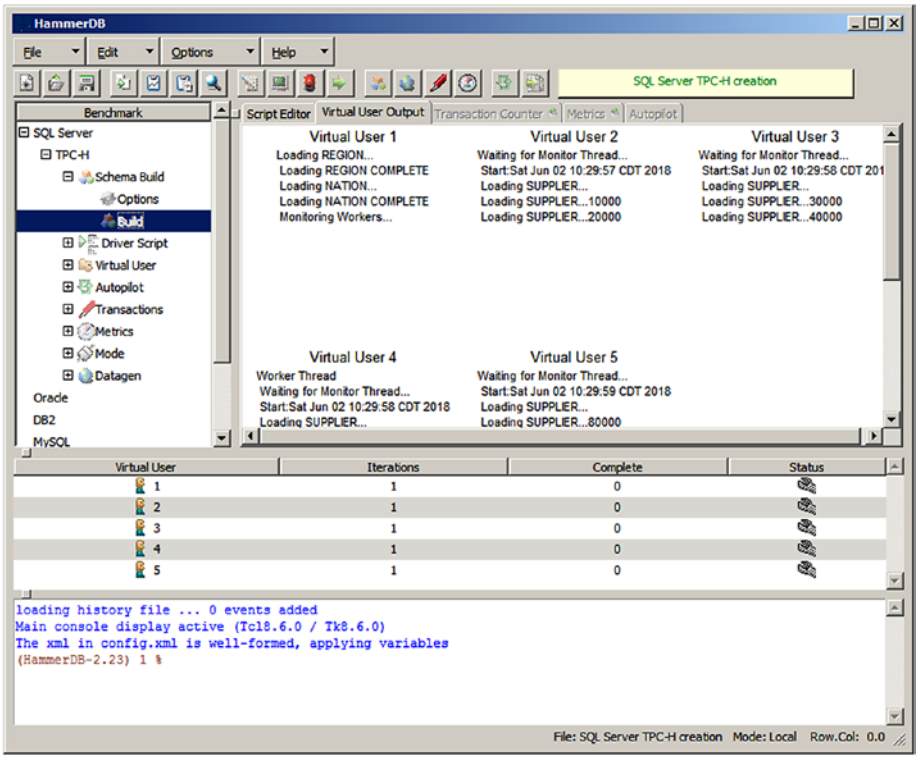


Figure 3-14. *Critical Build Schema Options*

There are two aspects of this window to focus upon: the upper right and middle sections. The middle is fairly simple: it's a display of the virtual users running, their status, and if they have completed. The first noteworthy item is that while we chose four virtual users there are, in fact, five' why? The answer is simple. When you ask for more than one virtual user then there is a coordinator/monitor user session, in this case virtual user #1. You can tell by looking at the first line for each virtual user, user #2

though #5 shows waiting on the worker thread, which means they are waiting on the controlling session.

You'll know when the build process is finally 100% when the display contains the four key items highlighted in Figure 3-15. First that the virtual user threads have all completed. Second that the status for each virtual user shows a green check mark. Third that each of the child virtual user threads ends with TABLES COMPLETE. Fourth and finally that the main virtual user thread shows CREATING INDEXES, UPDATING STATISTICS, and SCHEMA COMPLETE. Anything less and you will need to drop the database and its objects in order to repeat this step. Note that there are many things that can go wrong, so don't be surprised nor upset if you do in fact have to repeat this step.

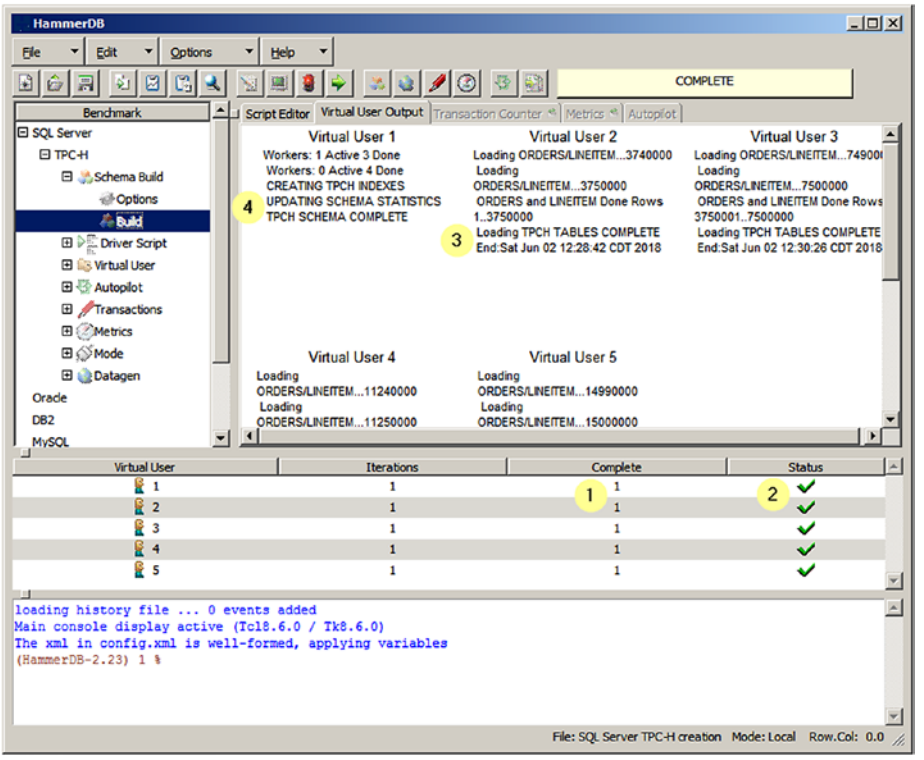


Figure 3-15. *HammerDB Load Complete*

Step #2: Load Driver Script

We now continue by expanding the "*Driver Script*" tree-view node and then double-clicking on the "*Options*" node, which launches the benchmark script coding options as shown in Figure 3-16. Note that step #2 as a whole is much simpler and thus shorter than the prior step; therefore this section is rather brief on purpose.

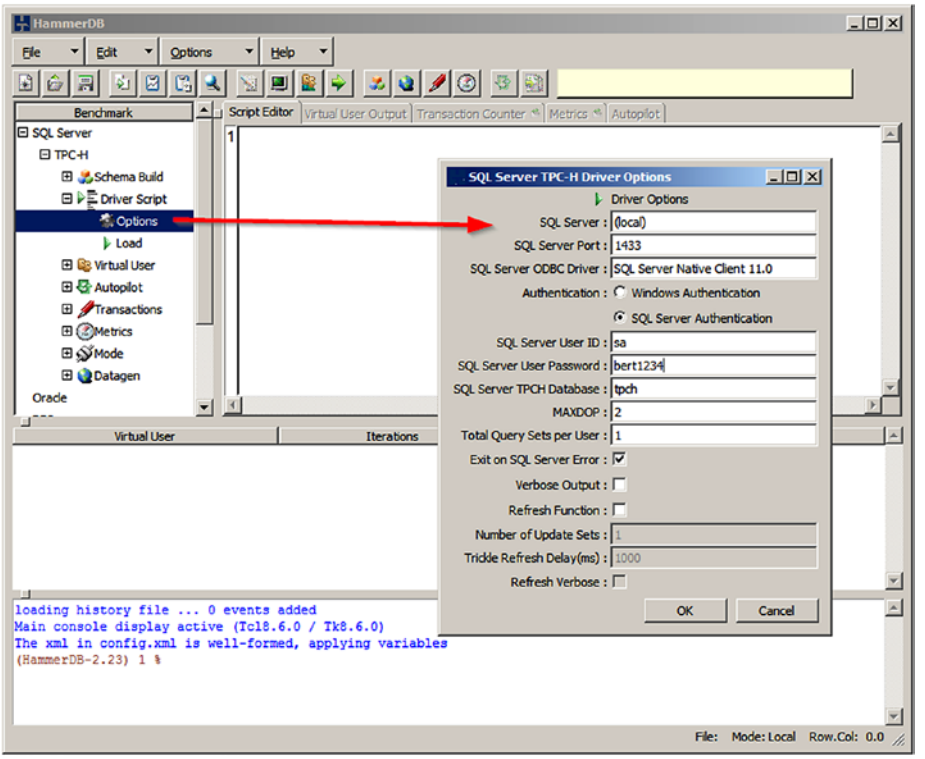


Figure 3-16. *HammerDB Driver Script Options*

Once again we simply define the database connection criteria, which database contains the objects in, the maximum degree of parallelism (MAXDOP), and how many iterations of the 22 TPC-H queries per virtual user. As before while this screen can vary by database platform,

most database administrators will find all of the requested information fairly obvious.

As with prior schema options screen, the driver options screen can vary slightly depending on the database benchmark being set up as shown in Figure 3-17. Note well that HammerDB defaults the "*keying*" and "*thinking*" times to unchecked (meaning zero time) even though required by the spec. I suspect that's because of the product's name, which contains the word "*hammer*" and so the default was chosen to favor hammering the database for the highest TPS possible. Yet other database benchmarking tools (e.g., Benchmark Factory) do not offer the ability to enable or disable these settings. I've seen many DBAs wonder why one tool produces radically different TPS and average response times due to their lack of attention to this one detail. They usually just assume that HammerDB is superior since it scored higher TPS. So make sure to pay attention to these types of settings across tools if you want to be able to make apples to apples comparisons.

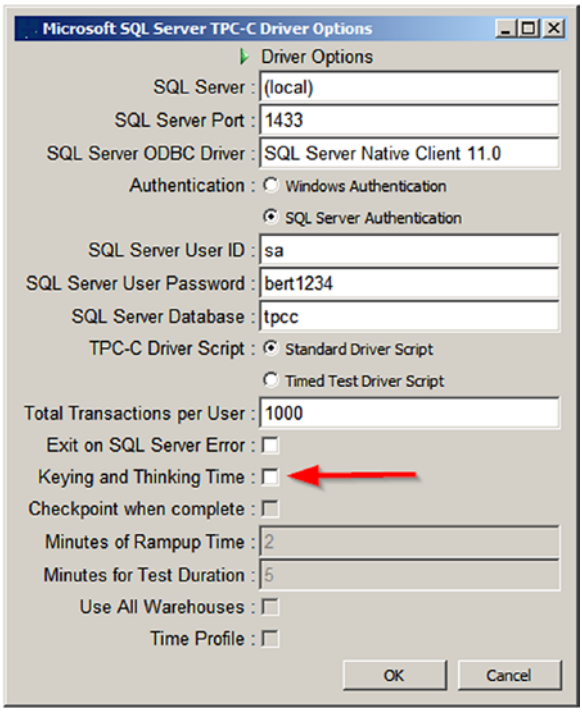


Figure 3-17. *Differing HammerDB Driver Options*

With the driver script options now selected, it's time to simply load the benchmarking script with those values reflected. For that we again look under "*Driver Script*" tree-view node, but now double-click on the "*Load*" node, which loads the resulting benchmark execution script into the script editor as shown in Figure 3-18.

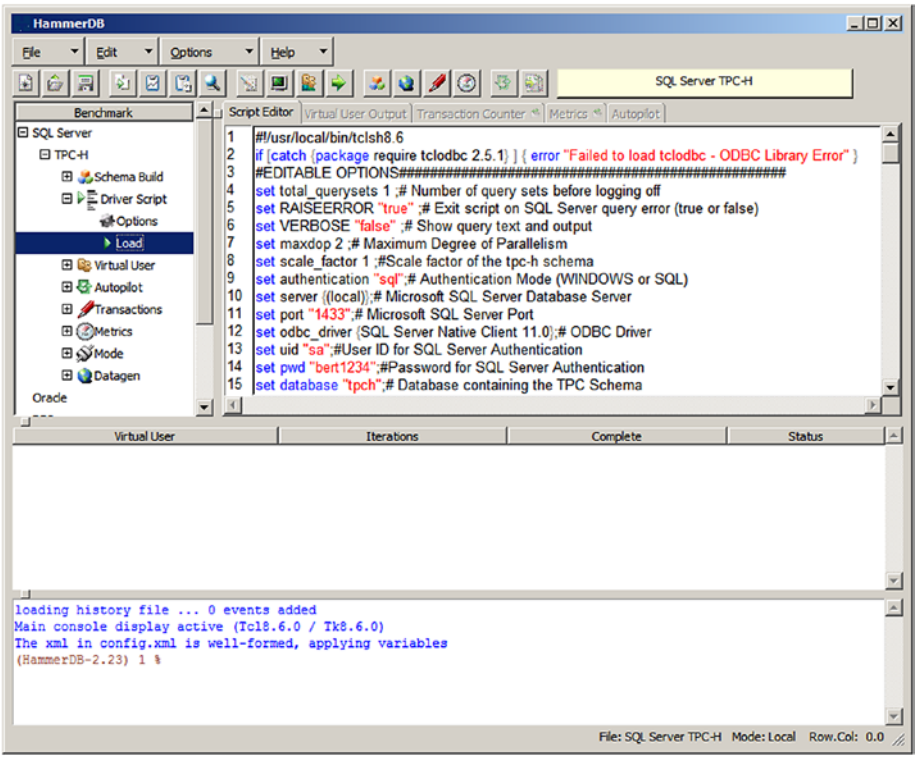


Figure 3-18. *HammerDB Resulting Driver Script*

I suppose that any competent TCL programming language coder could choose to modify the generated script. However, I've discovered that this TCL code attempts to implement the benchmarking spec and is thus fairly complex. So my recommendation is to just change options and reload the new resulting script. But I am not a programmer anymore so maybe I'm just being scared and lazy. I leave it up to the reader to decide if they can possibly improve upon the generated code. Just remember to be sure to save your modified script such that it's available for future use.

Step #3: Create Virtual Users

We now continue by expanding the "*Virtual User*" tree-view node and then double-clicking on the "*Options*" node, which launches the virtual users options as shown in Figure 3-19. Note that step #3 is also a shorter and less complex step; therefore like section #2 this section is rather brief on purpose.

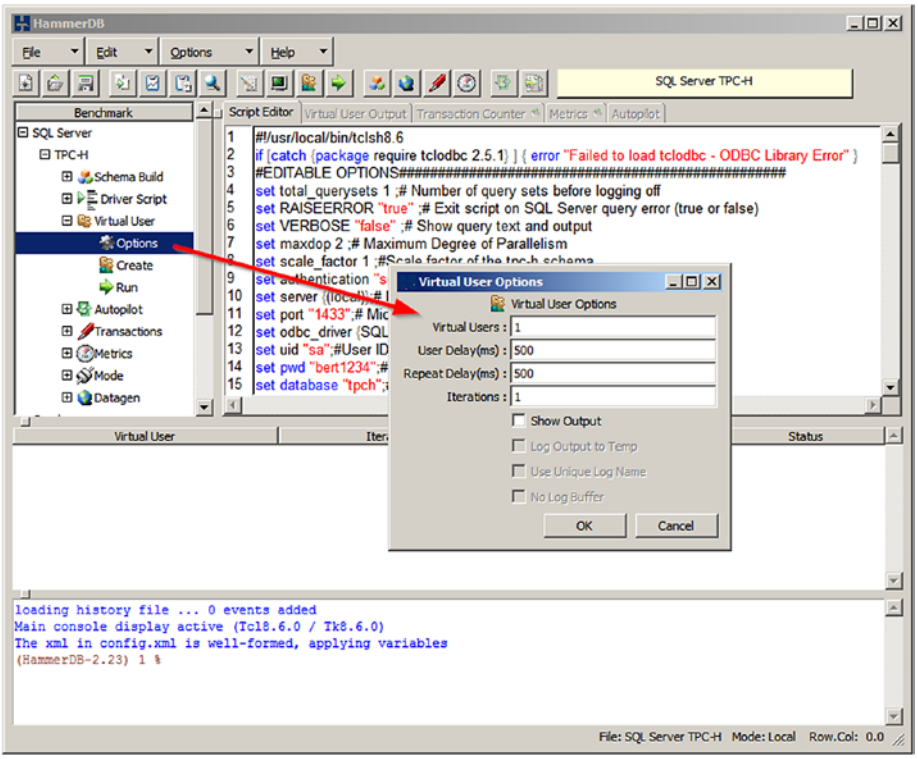


Figure 3-19. *HammerDB Virtual User Options*

Here we must specify the number of concurrent virtual users desired, length of virtual user delay or "*think time*," length of the processing delay or "*lag time*," and the number of iterations to repeat the entire workload per virtual user. Note that while this screen in Figure 3-18 seems pretty

straightforward and easy, you want to take your time on this screen as these values may be defined and/or mandated by the spec and not default appropriately on this screen. Failure to pay attention here could well result in unexpected and unrepeatable results.

For example, let's assume we had instead been instead doing a TPC-C benchmark and during the database object creation and data loading stage had requested, say, 10 warehouses. The TPC-C spec clearly defines the maximum allowable number of users per warehouse as 10. Yet this screen would allow us to request 200 concurrent virtual users and further would not warn that this is out of spec. This would likely result in undo database row contention and also incorrect results due to insufficient, separate rows for the users to process. You might even see database deadlock errors. No matter what, the results will likely be unreliable and unrepeatable.

There is also one more major caveat worth mentioning here. While HammerDB offers the ability to define both the delays between virtual users starting their processing and between iterations, other tools might not. Once again settings like this can make two different database benchmarking tools seem to yield different results. But the harsh fact is you reap what you sow. Therefore, pay attention to details like these.

With the virtual user options now selected, it's time to spawn those virtual user session processes as idle, but ready to go. For that we again look under "*Virtual User*" tree-view node, but now double-click on the "*Create*" node, which then spawns or forks those idle virtual user sessions as shown in Figure 3-20.

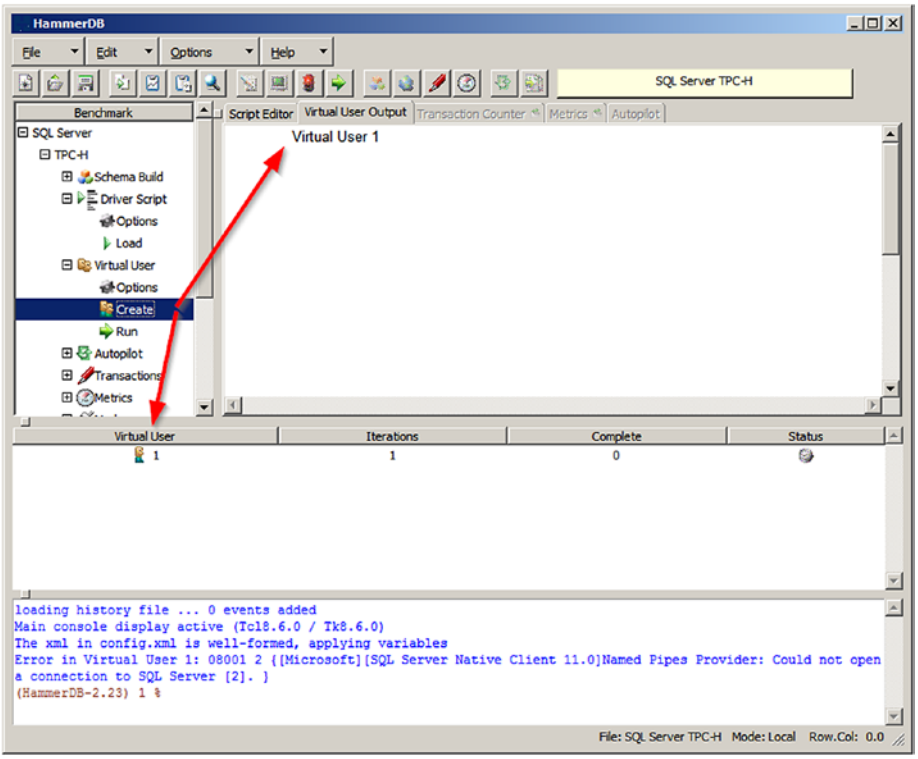


Figure 3-20. *HammerDB Virtual User Launch*

There are a couple of noteworthy items in Figure 3-20 to recognize and appreciate. First, there are exactly the requested number of virtual users, and there are no controlling or monitoring additions as there was in the database object loading stage. Second, if you have requested 100 virtual users, then make sure to scroll through the list and verify that you actually got the requested number spawned and ready to go. It's very easy to forget that step and not realize the reason the results are unrepeatable are that the virtual user count has varied due to some unforeseen issue such as insufficient memory on your machine or database limits on the number of concurrent connections allowed.

Step #4: Execute the Benchmark

We are finally to the most important stage: actually running the database benchmark. For that we continue by expanding the "*Virtual User*" tree-view node (with the assumption that the virtual users are already spawned) and then double-click on the "Run" node, which simply releases the idle virtual user sessions to each execute the generated script with the user-defined scripting options as shown in Figure 3-21.

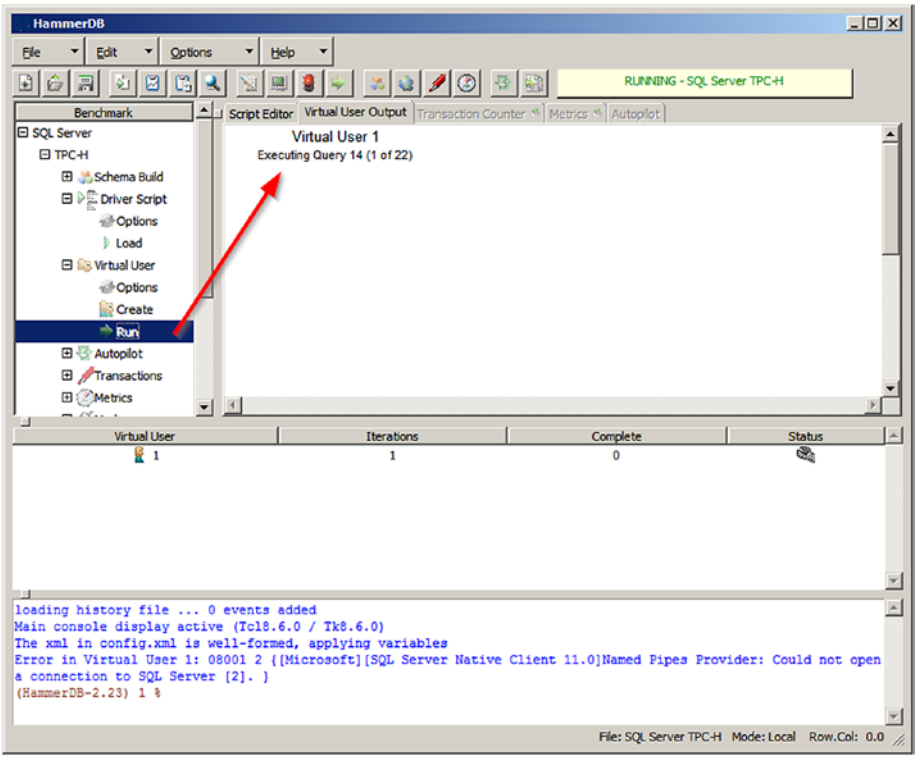


Figure 3-21. *HammerDB Executing a Benchmark*

Note that the output shows that the first query of the 22 defined in the spec being run is query #14. Don't let that confuse or bother you as it will run all the queries as required. The order is simply randomized such that not all concurrent sessions will run the same query at the same time. The random order will more closely match how real users would mostly all be running different queries at different times. You'll know when the benchmark has successfully completed when you see screen output like that shown in Figure 3-22. Note that the virtual user shows it completed with a green status and that it took 150 seconds to run the 22 queries.

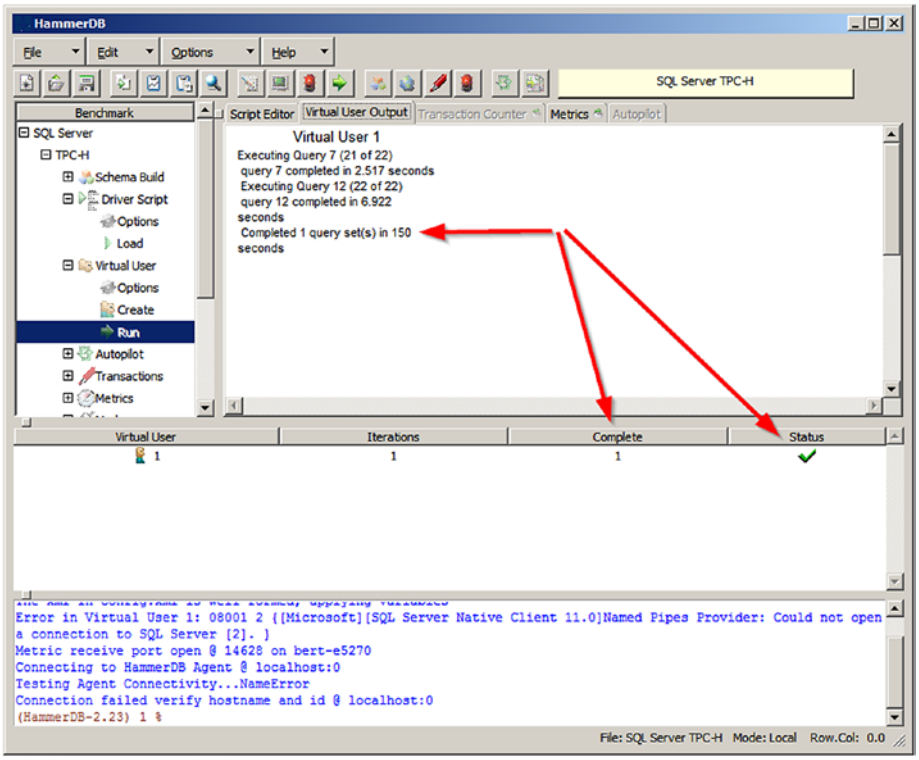


Figure 3-22. *HammerDB Benchmark Completed*

Monitoring the Performance

Note that simply knowing how long it took to run a benchmark is generally not the most robust and useful statistic. For the TPC-H the queries per hour (QpH) might be more useful, and for the TPC-C the transactions per minute (TpM). Neither of these is the final reported value that the spec requires be calculated for publication, but usually these are good enough for most DBAs. So how do we get such information from HammerDB? It's quite easy actually; we simply initiate the transaction counter. Many DBAs do this only during the execution portion of the benchmark, although you can monitor the create and load phase as well if you wish. Either way all you have to do is to define the transaction counter's database connection info as shown in Figure 3-23.

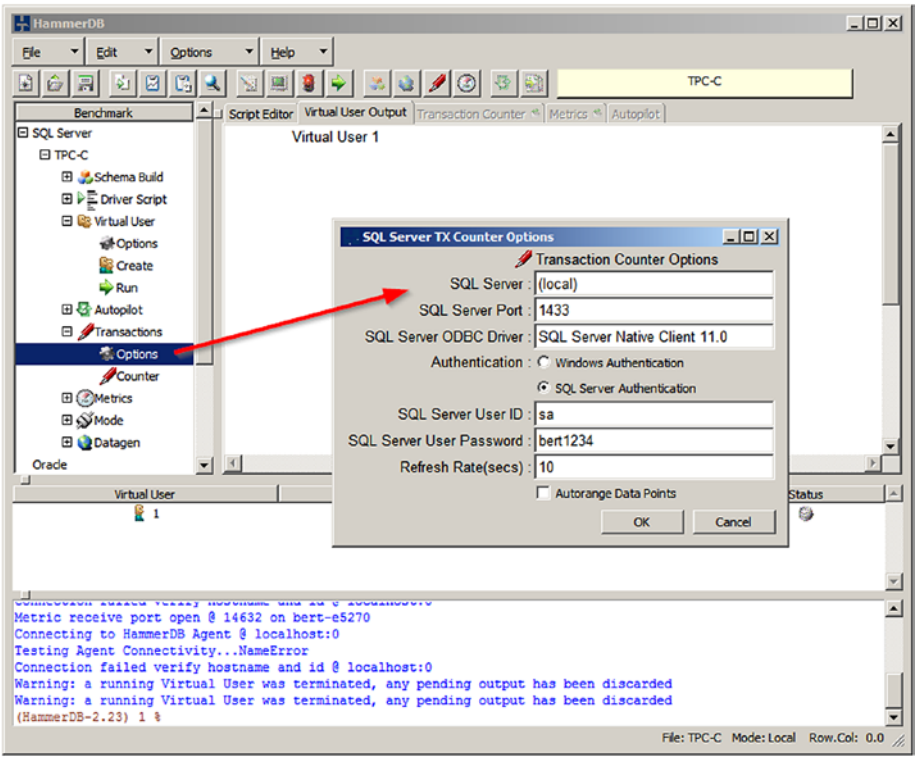


Figure 3-23. *Configuring the Transaction Counter*

Then you simply initiate the transaction counter collection by double-clicking on the "Counter" tree-view node, which results in the waiting status for the transaction counter screen as shown in Figure 3-24.

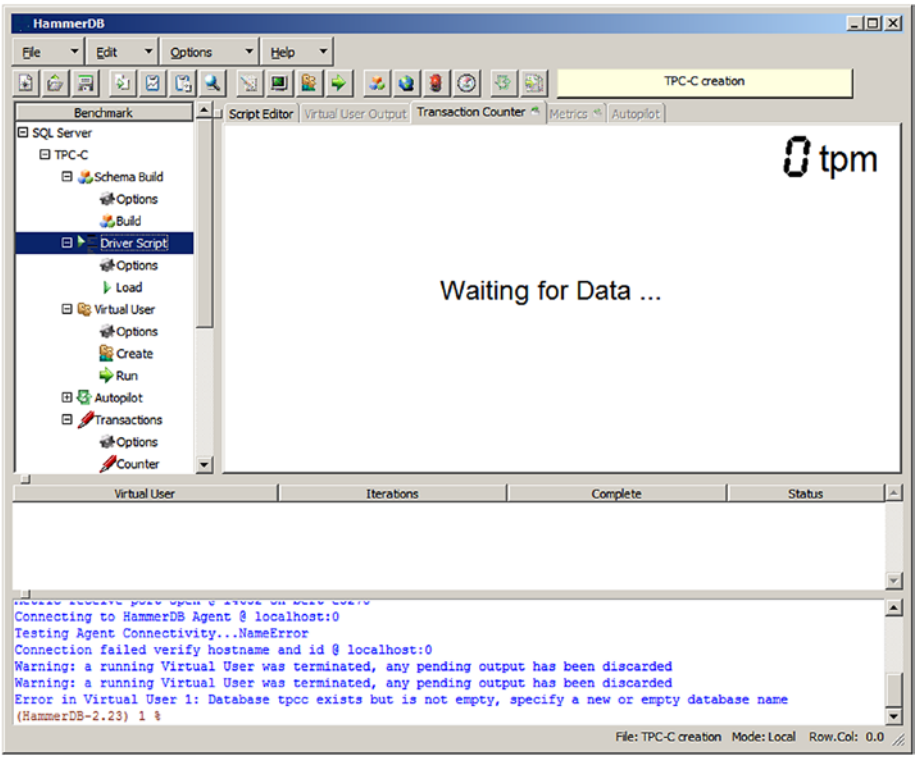


Figure 3-24. *Transaction Counter Waiting*

Now when you run a database benchmark you will see the transaction counter screen updated as shown in Figure 3-25.

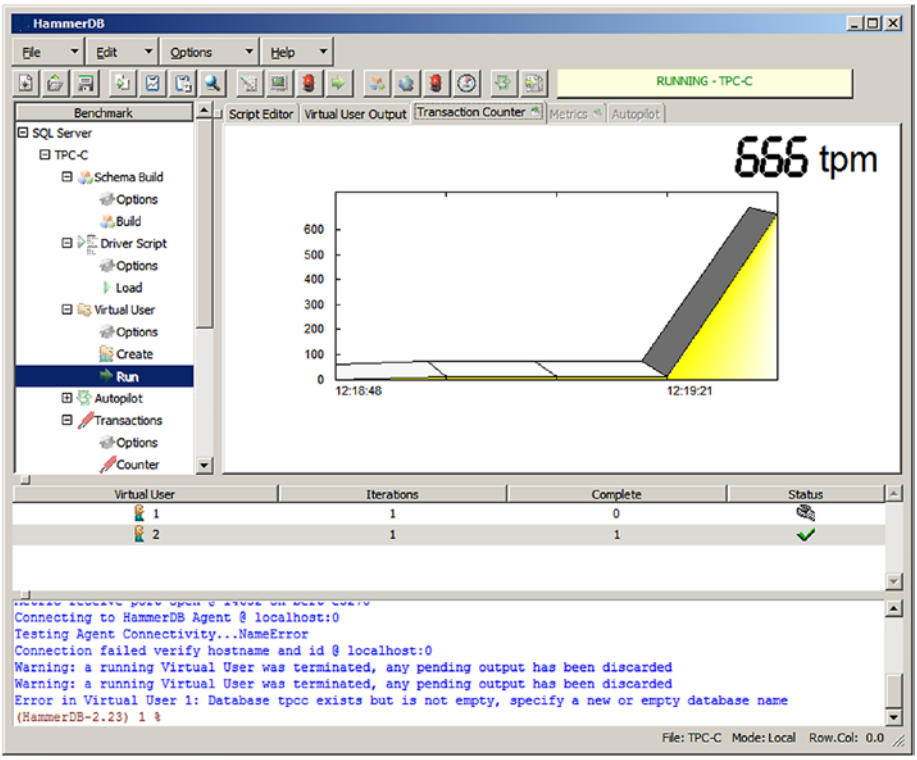


Figure 3-25. *Transaction Counter Working*

Series of Executions

There is one final trick to learn when using HammerDB, "How do we run a series of increasing virtual user counts in order to find the maximum value possible?" In fact, you might well argue that all of your benchmarking efforts will be done this way; therefore this is a critical aspect to learn and master. It's actually quite easy, after the database object and driver load steps you simply enable the Autopilot feature, specify the run time per iteration, and the number of virtual users per iteration as shown in Figure 3-26.

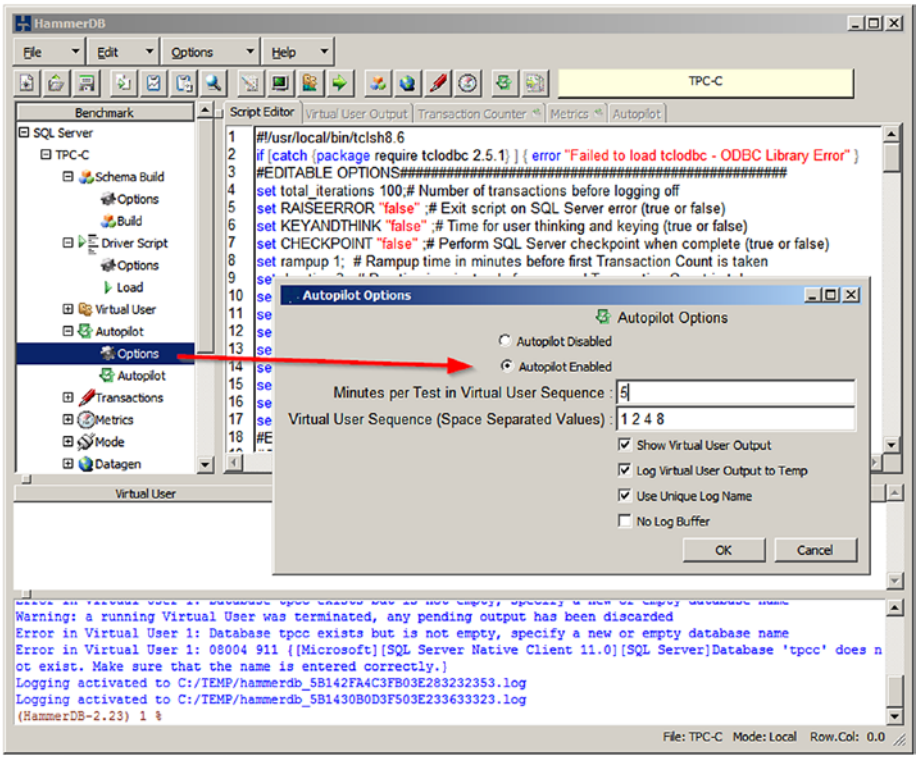


Figure 3-26. *Using HammerDB Autopilot*

In Figure 3-26, I have specified 1, then 2, then 4, and finally 8 virtual user scenarios. I cannot ask for more since I know that I created only one warehouse and thus cannot have more than 10 virtual users as per the spec. I also checked the boxes to show the virtual user output and to log each user iteration to a uniquely named file. Take special note that I specified a five-minute iteration period. How did I choose that number? Simple, it's based on what you specified for a timed run in the driver script as shown in Figure 3-27.

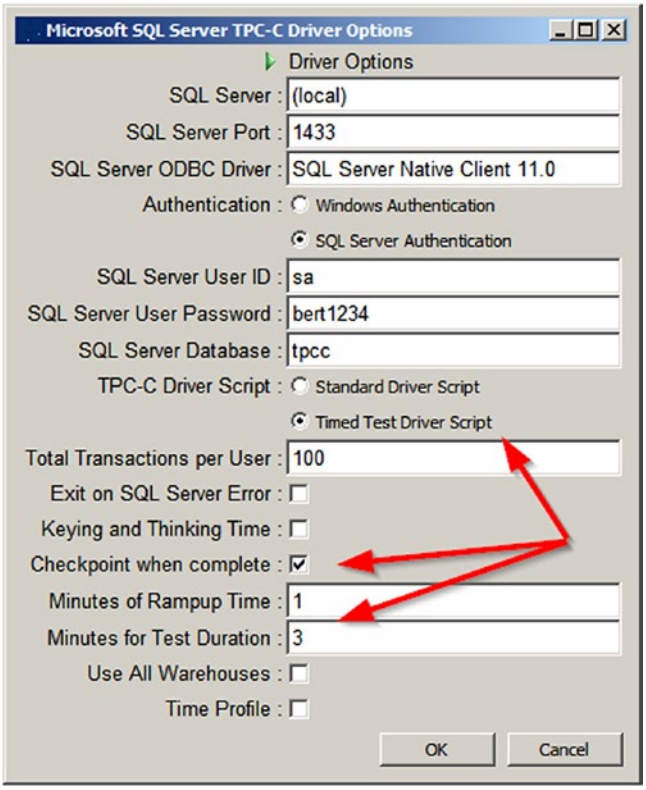


Figure 3-27. *Driver Settings for Autopilot*

First, you must choose the timed driver option. Next, the ramp up and test duration times must total less than the minutes specified per sequence back in Figure 3-26. Plus, it's highly recommended to have a checkpoint performed between each iteration to make sure that the database is in a consistent state at the start of each. Finally, note that this step must be chosen in lieu of creating virtual users and running the benchmark (i.e., steps #3 and #4 from earlier in this chapter) as all those steps now will be entirely automated.

Now we simply need to launch the autopilot as shown in Figure 3-28. Note that this process will take roughly 20 minutes to run as I have specified four iterations each of five minutes. You may have to experiment

with finding the right time period value as the ramp up time + test time + checkpoint time won't always be exact. My advice is to choose a value that's at least one to four minutes more than their total.

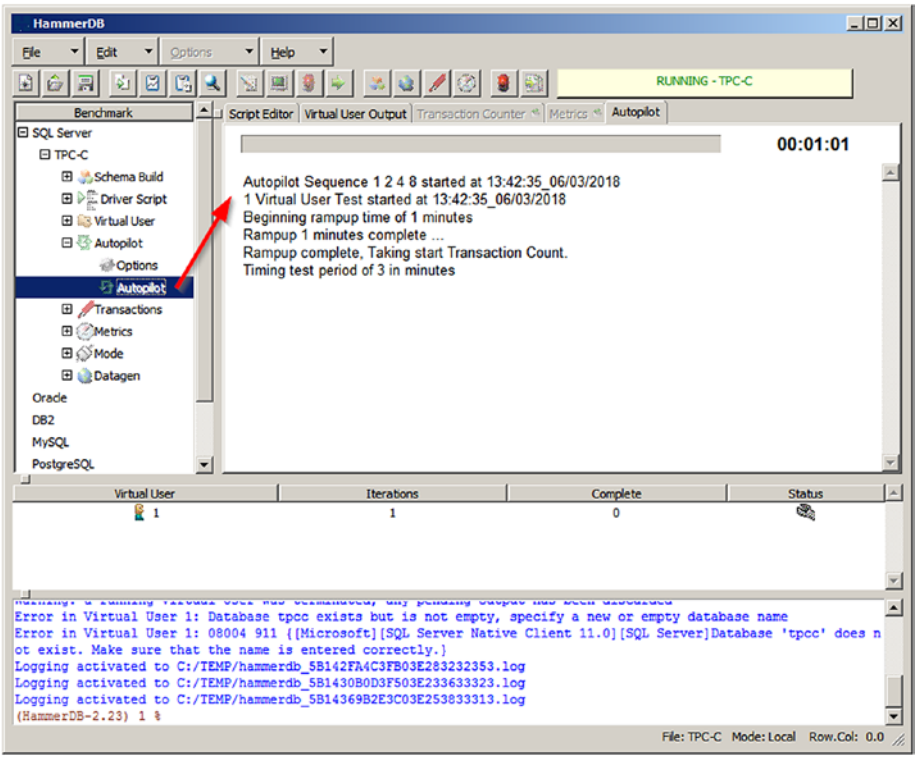


Figure 3-28. *HammerDB Autopilot Running*

When the entire test is finished running, you can then plot the TPM numbers for each iteration either from the log window in Figure 3-28 or fetch the values from the textual log files.

Benchmark Factory

Another popular database benchmarking tool that works with multiple database platforms is Quest Software's Benchmark Factory for Databases. While it is a commercial software offering (which of course costs money), there is also a very good freeware version (with a 100 concurrent database session limit). I have used this tool for many years and believe it to be a darned good database benchmarking tool for those wishing to quickly execute and score an industry-standard database benchmark. In that respect, it's very similar to HammerDB. However, unlike HammerDB whose user interface can be difficult to learn at first, Benchmark Factory offers very simple wizards to perform all the steps in HammerDB, which took the prior 20 pages in just a few simple steps. As such, this section will be far shorter than the last one.

You will find Benchmark Factory's web page, community, and freeware download at the following URLs respectively.

- https://www.quest.com/products/benchmark-factory
- https://community.toadworld.com/products/benchmark-factory
- https://www.toadworld.com/download/benchmark-factory-for-databases/freeware

Benchmark Factory offers only a Windows version for both 32-bit and 64-bit. The current version is 8.0. When you run the installer on Windows it will install the software under the **C:\Program Files** or **C:\program Files (x86)** directory as you would expect (based upon the bit size of the version being installed). Then to run the software you just double-click on the Benchmark Factory desktop or start menu icon. When the Benchmark Factory software launches, it displays the screen shown in Figure 3-29.

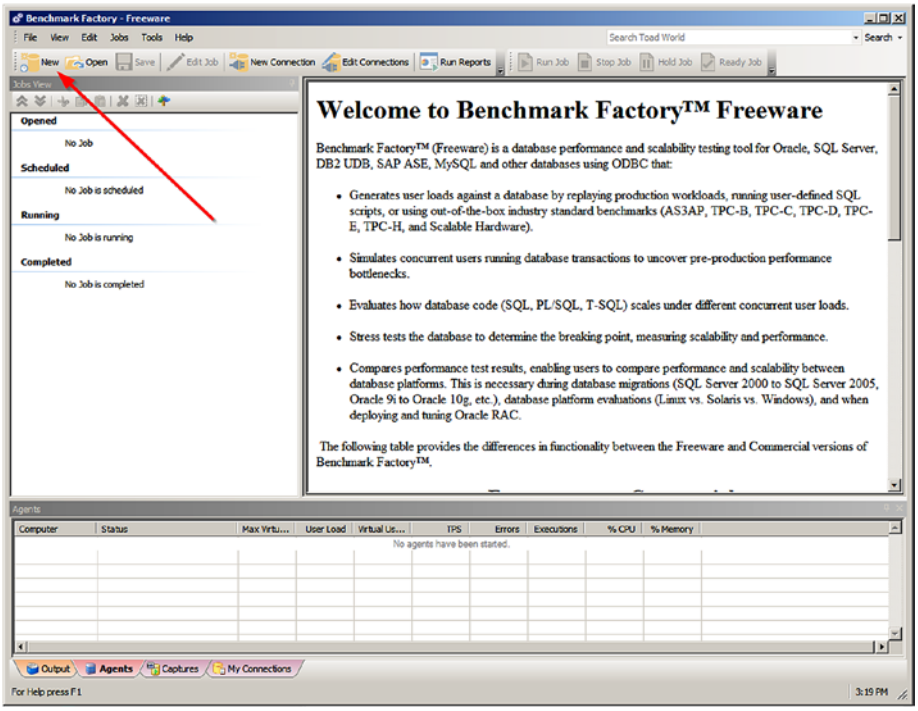


Figure 3-29. *Quest's Benchmark Factory*

In order to define and run a database benchmark you simply press the new toolbar icon that is highlighted in Figure 3-29. You can also get to this via the File menu. The first step in the wizard will define a database connection for running the benchmark as shown in Figure 3-30.

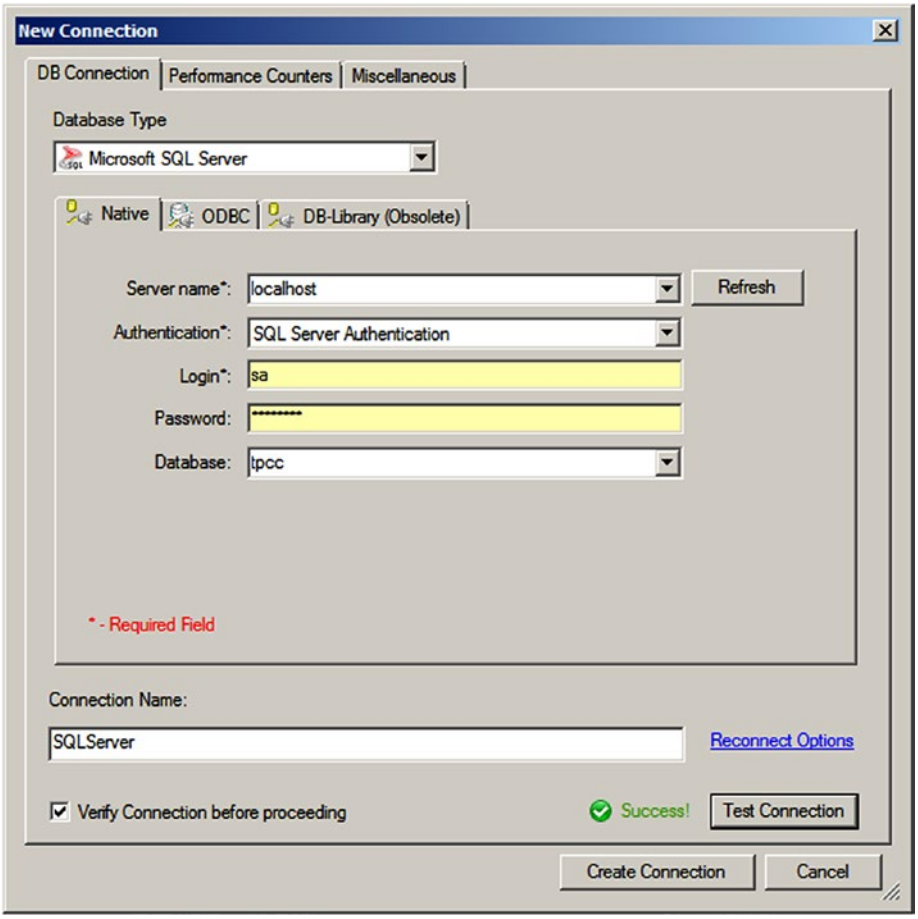


Figure 3-30. *Define Database Connection*

Had we instead chosen to run the TPC-C database benchmark on Oracle then the fields presented on this screen would have been slightly different. So even though this screen can vary by database platform, most database administrators will find all of the requested information fairly obvious.

The second step in the wizard is to select a database benchmark to run as shown in Figure 3-31. Note the drop-down box near the top that shows industry-standard benchmarks. Benchmark Factory offers several other

tests, including scalability tests, custom tests designed by the user from existing tests (e.g., 50% TPC-C and 50% TPC-H), and workload capture/replay.

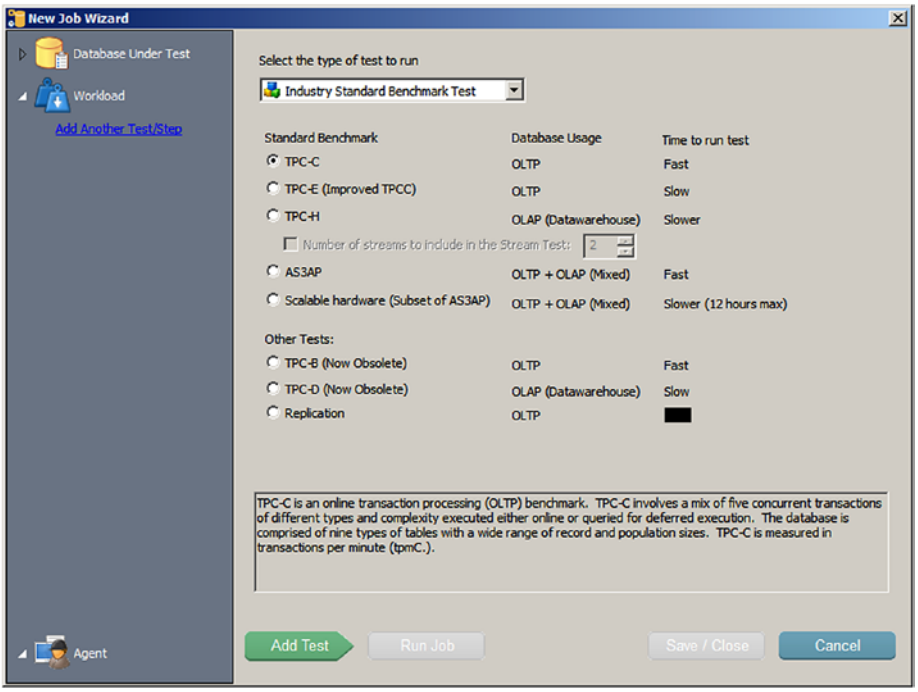


Figure 3-31. *Select Database Benchmark*

The third step in the wizard allows the user to define all the execution parameters and options as shown in Figure 3-32. Note that I've selected to run a TPC-C with a scale factor of two, which will consume 162 MB of space, and with user session iteration counts of 1, 2, 4, and 8 like was done in the prior section with HammerDB. All that's left to do is to press the run job button.

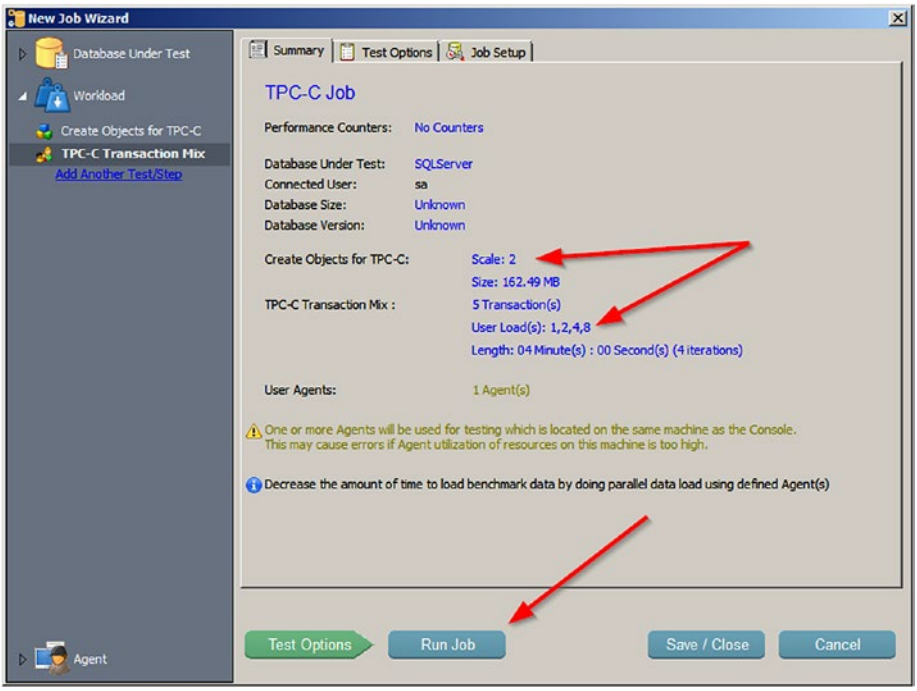


Figure 3-32. *Select Database Benchmark*

When the job starts running, it will first create and load the database objects as shown in Figure 3-33. Note that this will include creating all the required indexes, key constraints, and statistics gathering. Moreover, for some database platforms such as Oracle, additional options are available such as take an AWR or STATSPACK snapshot before and after the test.

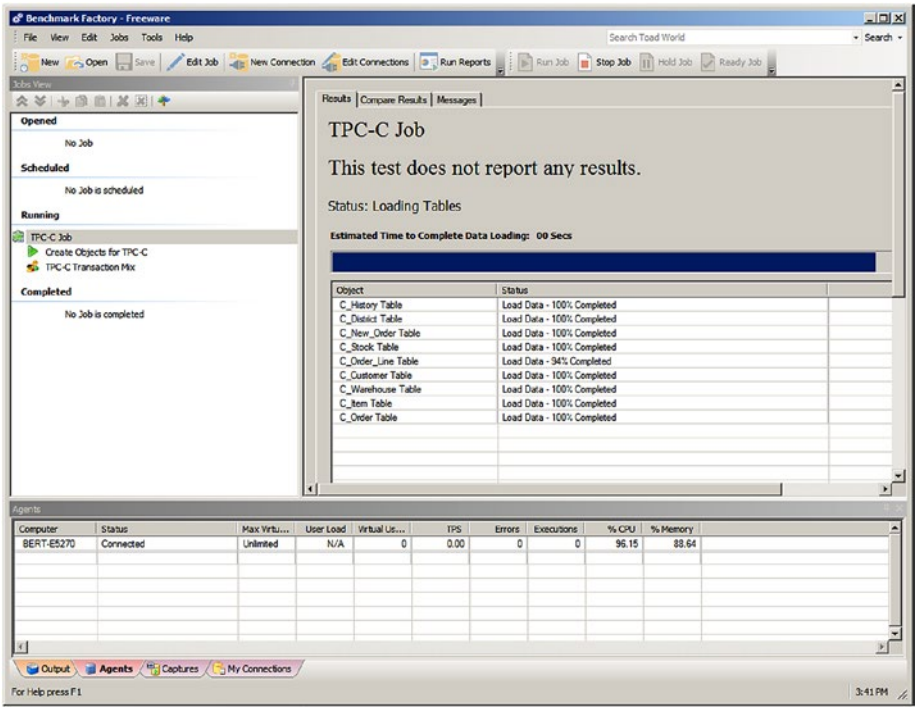


Figure 3-33. *Loading the Test Data*

After the data is loading, the benchmark runs as shown in Figure 3-34. Note how much more information it provides while running than HammerDB. There are nice progress bars for both the iterations to run and percent complete for the current execution. Plus, it has a nice graph that is updated every few seconds. Finally, it also displays the current values for both TPS and average response time. As such Benchmark Factory job execution is highly informative and worth watching as it runs for insights as to encountering performance limits.

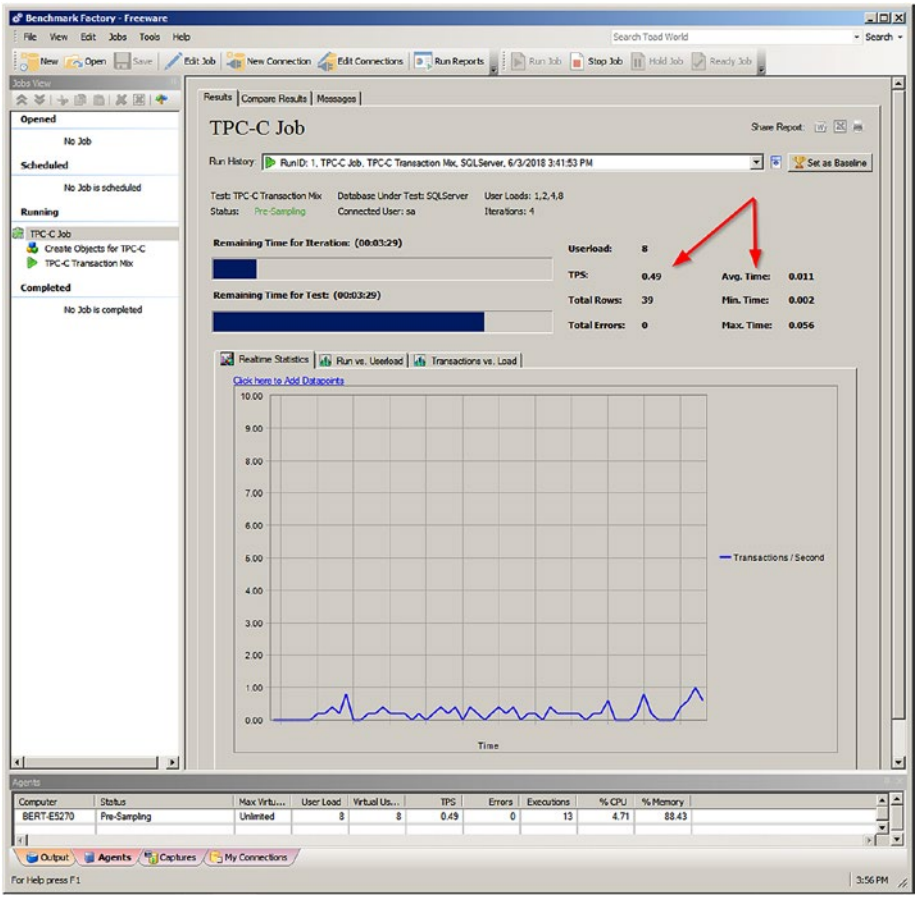


Figure 3-34. *Database Benchmark Running*

Finally, when the benchmark job has run to completion, Benchmark Factory presents a wonderful summarized view of the results as shown in Figure 3-35. Plus all that data in can easily be exported to an Excel or Word file. Plus you can set any test run as a baseline and then do a comparison with other test runs. In fact, the user interface is so functional that I have rarely had to export the data for any reason. In this area, Benchmark Factory excels.

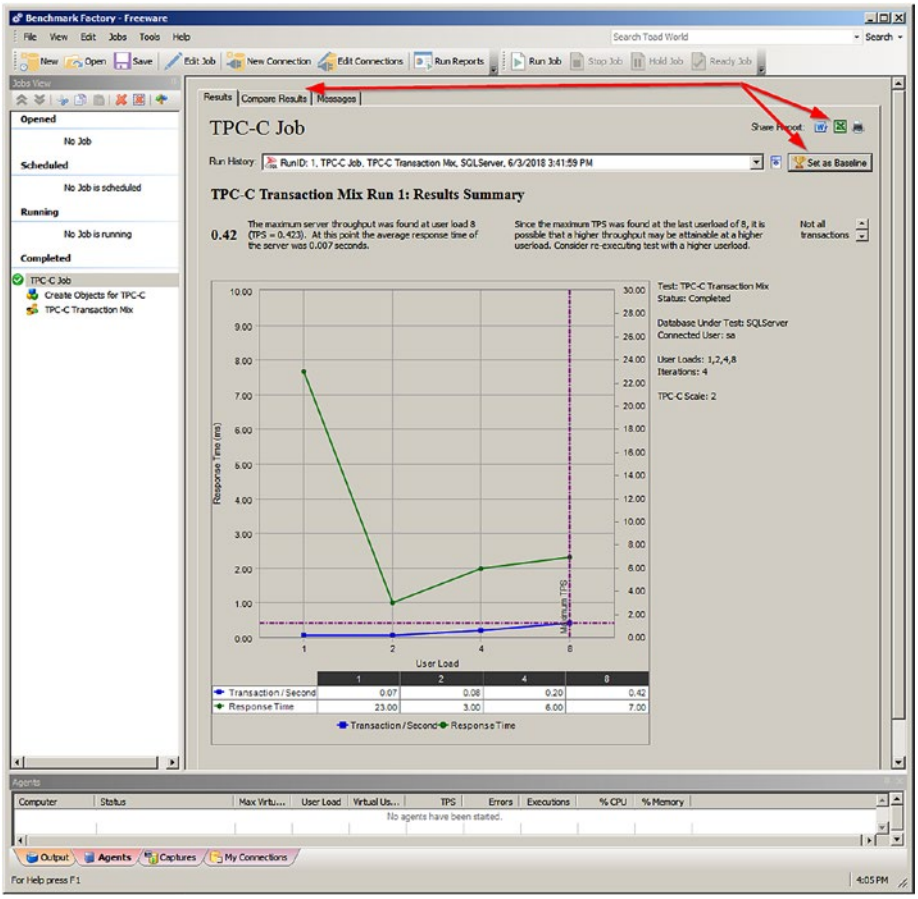


Figure 3-35. *Database Benchmark Results*

Swingbench

Another popular, free load generator and database benchmarking tool that works with just Oracle databases is Swingbench. I like this tool for one very simple reason: it's the only one I've found so far that offers the newer TPC-DS benchmark. In fact, this benchmark with its 100 complex queries is one of the best database performance tools that I know of (besides actual real workloads). My coauthors and I used this in our 2017 Oracle

book titled *Oracle Database 12c Release 2 Testing Tools and Techniques for Performance and Scalability*. In fact, the tool's author worked with us to make this book possible for which we are grateful (thanks, Dominic).

You will find Swingbench's web page and download at the following URLs respectively.

- http://dominicgiles.com/swingbench.html
- http://dominicgiles.com/downloads.html

The Swingbench download is a simple zip file that just needs to be unzipped. It's a purely Java-based tool so it can run on most any operating system; therefore it requires that Java be loaded on your machine. There are two types of executable files to be aware of: the roughly dozen utilities to load the desired workloads or benchmarks, and the actual Swingbench engine to run those tests. Here we'll just demonstrate running a TPC-DS benchmark. But the process is much the same for the other tests this tool offers.

Running the TPC-DS creation wizard walks you through creating a database connection, whether to drop or create the database objects, plus the owner and tablespaces for the required tables and indexes on the screen shown in Figure 3-36.

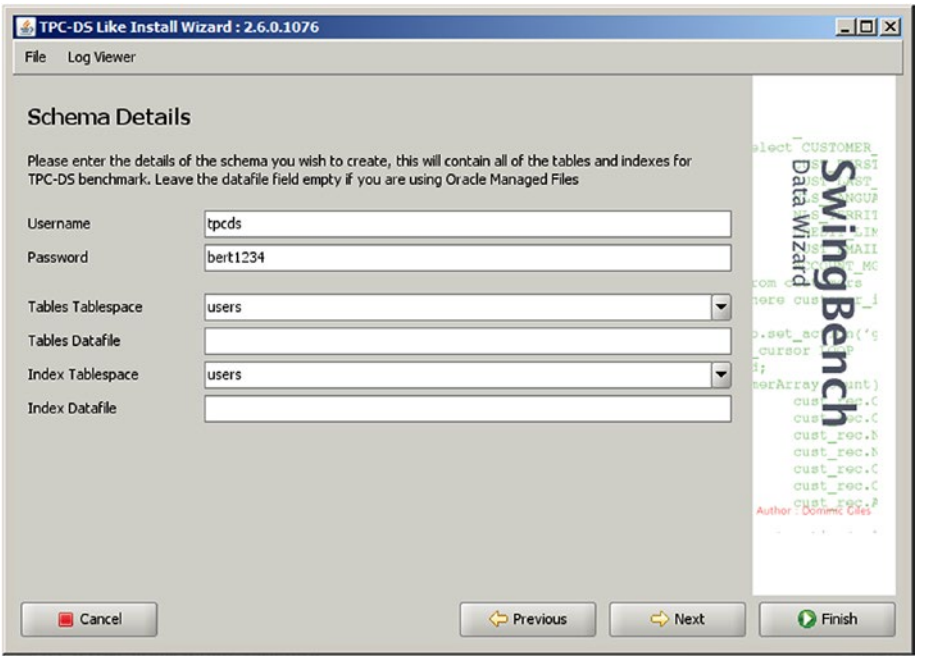


Figure 3-36. *TPC-DS Creation Wizard*

Along the way the wizard asks how big to make the database objects where each scale factor equates to 1 GB. The wizard then wraps up by asking the degree of parallelism to use while creating those objects. It then creates those objects as shown in Figure 3-37. Note that I have selected 8 parallel threads (which is the default) and currently the tool has loaded 13% of my data.

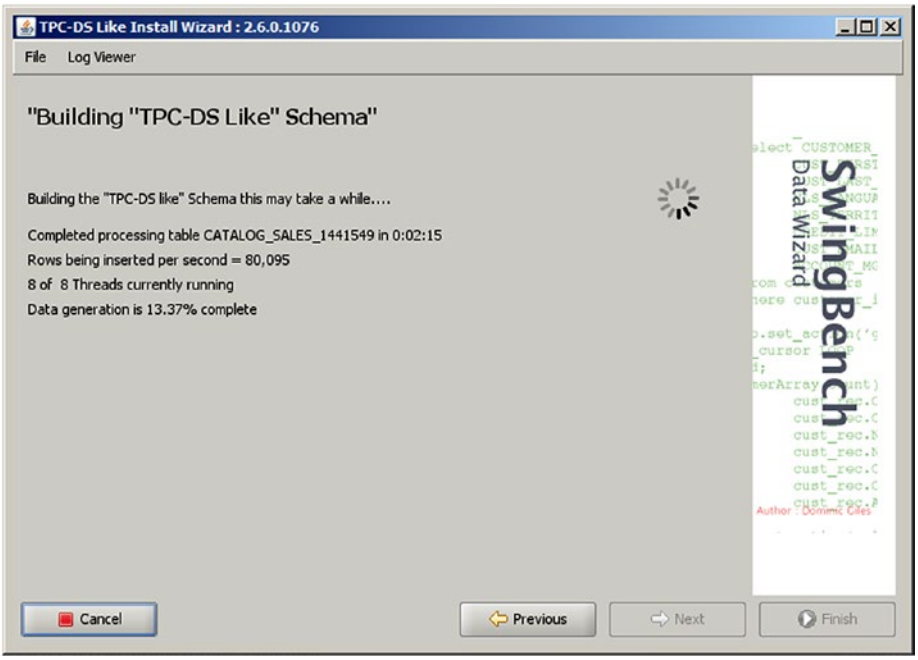


Figure 3-37. *TPC-DS Creation Running*

After the database objects have successfully been created and loaded, it's now time to run the Swingbench utility itself. When prompted during launch for what workload or database benchmark to run, choose the TPCDS-like test. Then simply enter the database connection info on the screen shown in Figure 3-38 and press the start execution toolbar button. Note that the "Load" tab in the bottom left of the main screen offers several key options that are quite important, including the min/max delays between transactions, logon delay time, whether to wait until all sessions connect before starting, and total run time. All of these can skew your results, so as always refer to the appropriate benchmark spec for advice and setting these parameters.

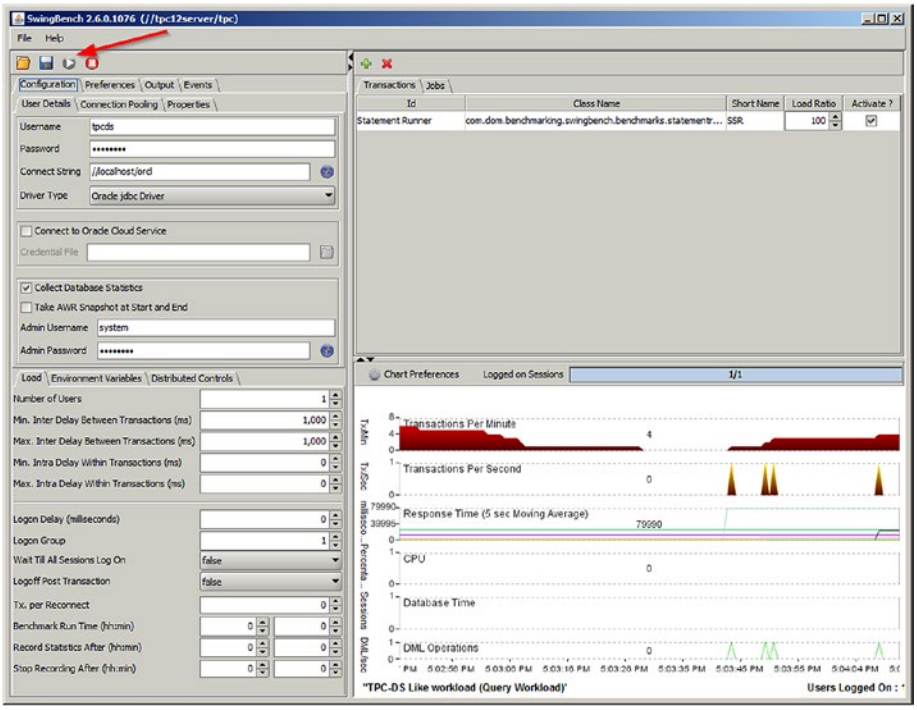


Figure 3-38. *TPC-DS Initiate Execution*

While the test is running you will see all the queries being processed (remember there are 100) and some key performance metrics as shown in Figure 3-39. I personally find all of the performance information that Swingbench presents to be the best of all the database benchmarking tools. Note that Swingbench shows two important graphs of note: the transactions per second and response time. As will be explained later in this book on how to interpret results, these are the two most important metrics to examine – and often you need to look at both as a whole rather than in isolation.

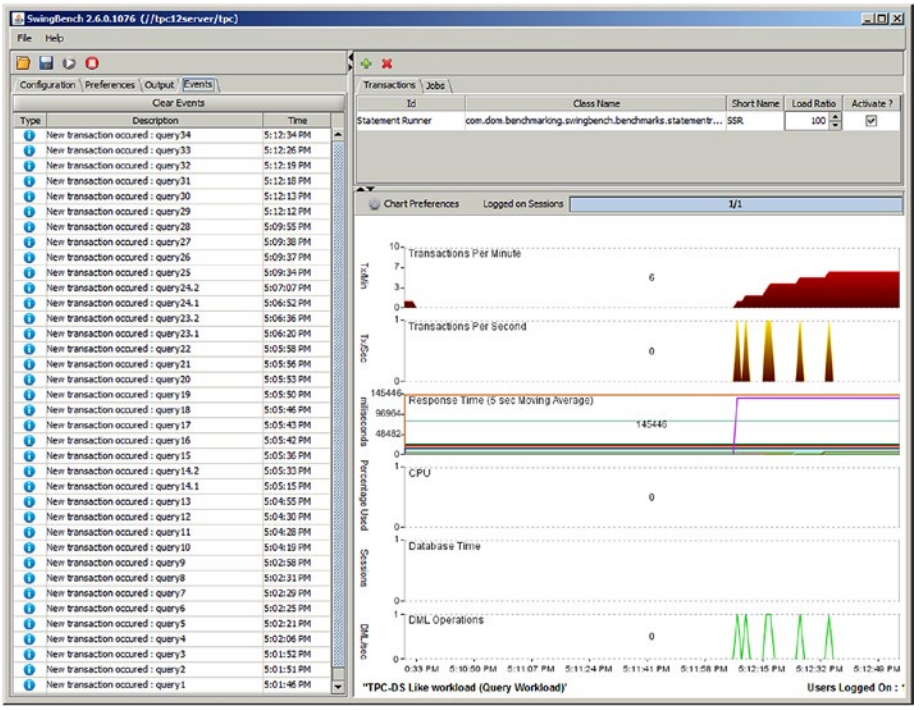


Figure 3-39. *TPC-DS Logging and Performance*

SLOB

One of the most popular and highly utilized tools for Oracle DBA's is SLOB, which stands for Silly Little Oracle Benchmark. It's also free so that tends to add to its popularity. I say highly used because almost every Oracle DBA under the sun knows about and uses it. But that does not necessarily mean that they appreciate its true purpose. To quote Kevin Closson, the tool's author:

> *SLOB is not a database benchmark. SLOB is an Oracle I/O workload generation tool kit. I need to point out that by force of habit many SLOB users refer to SLOB with terms like benchmark and workload interchangeably. SLOB aims to fill the gap between Orion and CALIBRATE_IO and full-function transactional benchmarks.*

Therefore SLOB really doesn't fit in the prior section dedicated to storage benchmarking, nor does it legitimately fit in here with the database benchmarking tools. But as I said, most Oracle DBAs utilize it, and even the tool's author admits they often misinterpret it as a database benchmarking tool. So we'll cover it here at the end of this section with those caveats.

You will find SLOB's web page and download at the following URLs respectively.

- https://kevinclosson.net/slob/
- https://github.com/therealkevinc/SLOB_distribution

SLOB is a collection of Linux shell scripts and some C programming language-based programs. There are two scripts you need to be aware of to use SLOB: setup.sh and runit.sh. There is also a text file named slob.conf that contains various parameters that control SLOB's runtime behavior. You will most likely need to edit this file for your environment. In my case I changed the SYSTEM password and the snapshot mechanism to AWR.

The first script, setup.sh, takes two input parameters: the tablespace where to create its schemas and their objects, plus the number of schemas to create. In my case I'm running a Linux virtual machine on my laptop so I want a minimal setup: use my exiting USERS tablespace and load just one schema. Figure 3-40 shows the execution of setup.sh with those values.

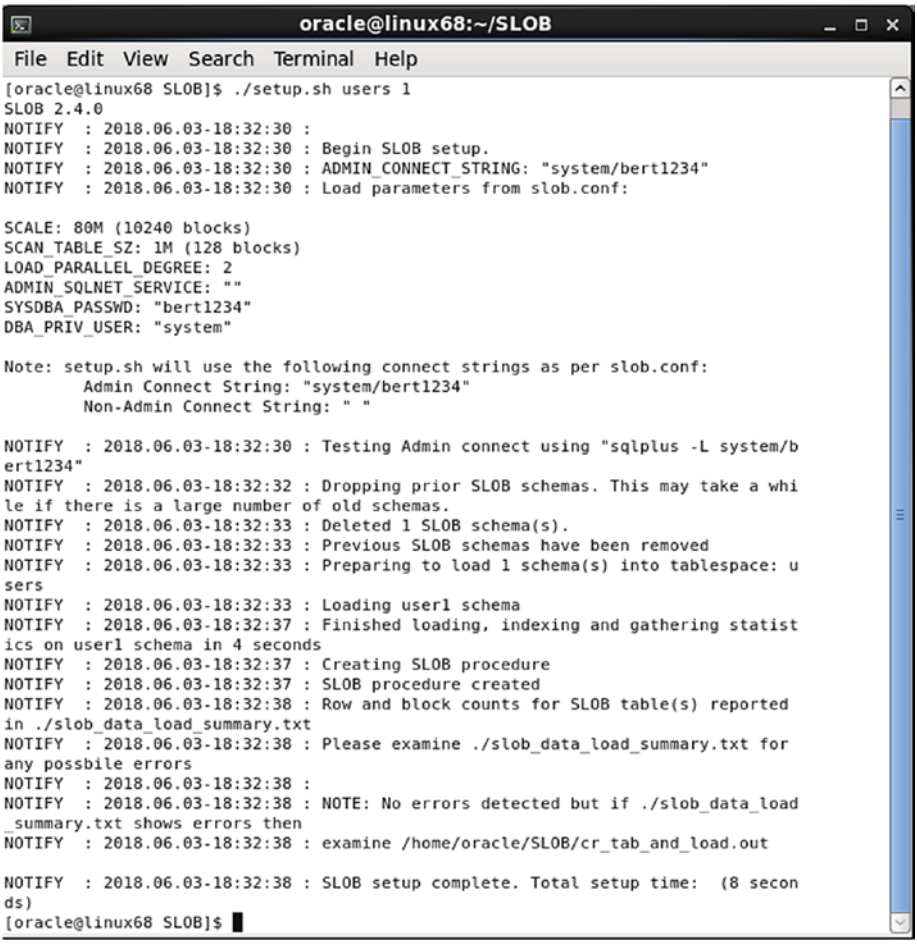


Figure 3-40. *SLOB setup.sh execution*

Once setup.sh completes with no errors, then it's time to execute the second script, runit.sh, which takes just one parameter: the number of schemas to test (which should be the same or less than the value you gave for setup.sh).

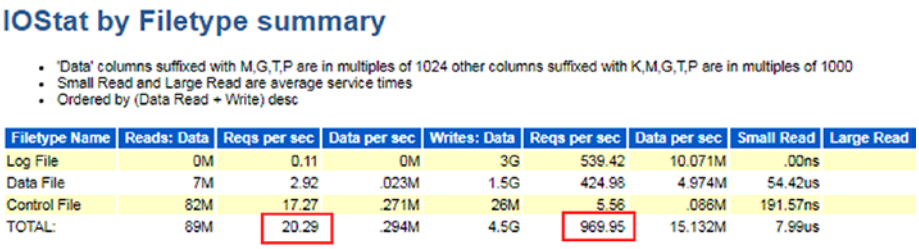
Once the job execution completes. then run an Oracle STATSPACK or AWR report for the begin and end snapshots listed by runit.sh output. Then examine the report concentrating on load profile section for the physical IO's quoted. My results are shown in Figure 3-41.

Load Profile

	Per Second	Per Transaction	Per Exec	Per Call
DB Time(s):	1.0	0.0	0.00	1.41
DB CPU(s):	1.0	0.0	0.00	1.34
Background CPU(s):	0.2	0.0	0.00	0.00
Redo size (bytes):	10,016,307.1	17,546.5		
Logical read (blocks):	374,342.0	655.8		
Block changes:	73,496.2	128.8		
Physical read (blocks):	2.5	0.0		
Physical write (blocks):	635.7	1.1		
Read IO requests:	2.5	0.0		
Write IO requests:	424.3	0.7		
Read IO (MB):	0.0	0.0		
Write IO (MB):	5.0	0.0		
IM scan rows:	0.0	0.0		
Session Logical Read IM:	0.0	0.0		
User calls:	0.7	0.0		
Parses (SQL):	4.8	0.0		
Hard parses (SQL):	0.5	0.0		
SQL Work Area (MB):	0.1	0.0		
Logons:	0.3	0.0		
Executes (SQL):	5,723.9	10.0		
Rollbacks:	0.0	0.0		
Transactions:	570.8			

Figure 3-41. *IOPS from AWRLoad Profile*

So according to most people who run SLOB and then read the AWR report's Load Profile section to calculate the IOPs, my laptop was running around 428 IOPS. However, I think that this number is not 100% accurate. I prefer to instead look further down in the AWR report as shown in Figure 3-42 at the IOStat by Filetype Summary, which shows the total to be more like 990 IOPS.



IOStat by Filetype summary

- 'Data' columns suffixed with M,G,T,P are in multiples of 1024 other columns suffixed with K,M,G,T,P are in multiples of 1000
- Small Read and Large Read are average service times
- Ordered by (Data Read + Write) desc

Filetype Name	Reads: Data	Reqs per sec	Data per sec	Writes: Data	Reqs per sec	Data per sec	Small Read	Large Read
Log File	0M	0.11	0M	3G	539.42	10.071M	.00ns	
Data File	7M	2.92	.023M	1.5G	424.98	4.974M	54.42us	
Control File	82M	17.27	.271M	26M	5.56	.086M	191.57ns	
TOTAL:	89M	20.29	.294M	4.5G	969.95	15.132M	7.99us	

Figure 3-42. *IOPS from AWR IOState Summary*

Summary

In this rather lengthy chapter, we covered all the well-known and popular IO subsystem stress testing and database benchmarking tools. For the latter, we covered all the steps required to run a database benchmark. Some tools were far easier than others. Some tools offered just one benchmark while others offered several. But no one tool currently offers all the industry-standard database benchmarks. So as a database benchmarking professional, you will likely need to master several of these tools.

CHAPTER 4

Benchmarking Preparation

At this point you may believe yourself fully ready to start your database benchmarking efforts, armed with just the information provided thus far: the database benchmarking basics, an overview of the various industry-standard database benchmarks, and an elucidation of the IO subsystem and database benchmarking tools available. Therefore, some of you might feel like this chapter is merely filler or fluff. But I'm going to impress that this chapter is, in fact, the most critical one in the entire book, and should be read with the most attention. In fact, I will further state that failure to prepare as described in this chapter is arguably the single largest contributor to the next chapter: missteps and failure.

When I was a young man I belonged to the Boy Scouts of America. While I may not have made Eagle Scout or earned lots of badges, I did learn quite a lot. The one thing permanently etched into my mind from then is the Boy Scouts motto: Be prepared! There is great wisdom in those two simple words. Taking that concept further, Benjamin Franklin said, "By failing to prepare, you are preparing to fail." Those words ring true for many things in life, but especially database benchmarking. Imagine an American football (not that soccer stuff) coach who does not prepare a game plan, would you really be surprised when they lost? In fact, you might join other fans in asking for a new coach if the current coach entered

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_4

a game with no plan or even just a poor plan. What's good for the coach is good for a database benchmarking professional. If ill prepared, then probably it is time for a new person.

Benchmark Preparation

As with any business venture wanting the best chance to succeed, the first thing to clearly identify and articulate is the mission statement for whatever database benchmarking efforts are to be made. Way too often this step is either skipped or simply marginalized. I know of no company whose mission statement is just "To make money." Likewise, most competent football coaches would probably not say that the game plan is to win just for the sake of winning (although with the notable, understandable, and entirely permissible exception of an Ohio State coach who must always beat Michigan just because they are Michigan).

So what in business constitutes a good mission statement? You will find many opinions and statements on this question. For our purposes, the four that are most applicable and important include:

- Creates expectations
- Goals genuinely realistic
- Is judiciously adaptable
- Is short, but with a clear purpose

Let's dissect this example mission statement along those four criteria:

> *To verify that moving our largest data warehouse and decision support databases from on premiss servers and a storage area network (SAN) to the cloud will perform just as well, for reasonable if not better overall costs.*

Wow – now that's seems like a respectable mission statement. In fact, it may well seem very similar to those thoughts that your management is already considering or will be asking you to address. In fact, over just the past 12 to 18 months, all the database benchmarking projects that I've been involved with have more or less been along these lines. It seems to be a very common theme these days.

Before we begin to dissect this mission statement, let's first agree that there are both technical and nontechnical aspects to it which might require a separation of responsibilities. Most of the DBAs that I work with are really only responsible for technical aspect – that is, at what cloud configuration (sizing) does it perform as well or better. That's not to say that the other part is not just as important, because it deals with costs. But most DBAs are best suited for just the technical aspect. So let's now only dissect the mission statement minus the seven words related to costs.

The expectations seem fairly straight forward; can the cloud perform as well as current systems? But there are several important questions buried in the phrase "largest data warehouse and decision support databases." Just how big is big? Are the DW and DSS workloads mostly complex queries, with some batch loading during off hours? Are end-user queries mostly from pre-canned reports or ad hoc, user-defined reports? Are we simply moving our current relational database such as Oracle or MS SQL Server to the cloud, or are we moving to a NoSQL database such as Hadoop or Cassandra? If there is a BI (business intelligence) server, will it remain on premises, or is it moving to the cloud? Now based on all these questions, you might very well assume that I'm going to say that this example mission statement missed on expectations. But if it's written in a way that induces one to ask such questions and to obtain useful answers, then I'd say mission accomplished (at least for the first bullet item).

Now on to the goals. Once again, we have a portion of the example mission statement that raises many questions: Where does "move to the cloud" mean and what does "perform just as well" mean? Are we moving to Amazon AWS, Microsoft Azure, Google Cloud, or Oracle Cloud. Do we

know what type and size virtual machine images we're permitted to use? Are we free to use NVMe and SSD, or are we restricted to standard storage disks? How are we going to test performance? Will we copy the database to run the application in parallel on the cloud? Or will we choose some industry-standard database and/or application benchmark and run that in the cloud? What is the current service level agreement (SLA) and what, if any, differences must be met for the new SLA? For example, most cloud vendors offer 99 point nine with either four or five repeating nines of uptime (e.g., 99.9999% vs. 99.99999%). That level of reliability probably is better than the current on-premise SLA, so do we have to meet the increased SLA across the board? All these questions actually relate to just one issue, how can we judge pass vs. fail? You may have noticed that once again the mission statement has raised more questions than answers. There is great value to this. This chapter is all about preparation. If the mission statement induces answers to all these questions before we start benchmarking, then we'll waste less time and increase our chance of success.

What about the adaptability of this mission statement? Getting answers to all the questions raised thus far should naturally raise issues of lead way to revise or adjust the approach. For example, if you are told it must be Amazon AWS and the initial thought was image types of "*general purpose*", are you permitted to suggest either "*memory optimized*" or "*storage optimized*" since those might better fit the requirements of a database server? The question of adaptability becomes critical when it comes to time estimates. If you are told no, the image types must be general purpose, then you can reasonably expect the schedule to be somewhat fixed. If on the other hand you are told to discover the best image type starting with general purpose, then your schedule just got expanded – possibly even doubled or tripled. Want about are there other hidden aspects worth inquiring about as part of adaptability? For example, maybe we know that the current application using the on-premise database does not in fact leverage any of the "*enterprise edition*" features. Therefore, should we perform testing to verify if a lower, less expensive edition of the database

will suffice in order to lower licensing costs? So yes, even more questions that require yet more answers. But that's the very definition of being prepared. In other words, "*look before you leap*," lest you suffer a similar fate as the goat in Aesop's fable who jumped into a well because the fox told him there was a drought coming but who just wanted something to stand on to get out of the well. You will be well served to nail down the allowed adaptability since it often simply represents scope creep.

The last bullet, is the mission statement short with a clear purpose, is seemingly the easiest part to digest. What is the "*big picture*" concept and the rationale for doing so? In this example, it's clear: to move to the cloud without any negative impacts. However, I once participated in a big large database benchmarking project where the company wanted to know which hardware vendor ran their preferred database the best. So the actual purpose was really who would get orders for some 10,000 servers over the next 5–10 years. That fact placed a greater onus on us to make sure that we were comparing apples to apples, and that there was no bait and switch shenanigans. This true underlying motivation clearly changed both our perception and approach to the project. While this effort may have cost only $20,000, nonetheless it was a mission-critical, strategic issue that rose to the most senior management levels. Armed with knowledge of this fact, we were able to provide a clear finding that we could justify and prove. So sometimes it pays to play dentist and to pull teeth, because at times what seems easy and innocuous is in fact a major issue if you just dig a little below the surface.

By now it should be clear, the mission statement should be your script from which to ask tons of questions in order to construct and verify a checklist that will become a key factor in resource allocation, scheduling, and budget for the benchmarking project. His checklist will also assure the best chance for success. Yet this critical step is almost always skipped, resulting in missed deliverables and time overruns. So I cannot stress enough just how critical this step really is. It's probably the single biggest factor in achieving success.

Database Preparation

The one area that shocks me the most when working with companies doing database benchmarking projects is often the lack of database preparation. In fact, it occurs more times than I would have ever thought possible. Plus there are multiple ways in which it can exhibit itself.

On enough occasion to warrant listing it, I have seen benchmarking projects being performed by business analysts or application project leads. I cannot explain the first scenario, but it happens. The second scenario is a little easier to understand; the application is what the end user sees and feels, so it's the application team that sometimes gets asked to test the effect of moving the database application from one deployment model to another. Either way the net effect is that non database administrators are asked to do a job where some of the most critical requirements are simply in a different resource's skill set. I have also seen where the application development team has a DBA, but they are an application DBA whose job it is to design tables and indexes needed by the application. They often are not experts in detailed performance analysis and optimization techniques. Regardless who is tasked to perform the database benchmarking project, they should include a seasoned DBA with skills on the target database and platform. I will even go so far as to suggest that the failure to include a competent DBA will almost universally guarantee project failure.

Another surprising oddity is the fact that about 50% of the database benchmarking projects I have been involved with simply install the database with preconfigured defaults across the board. In some cases, because as stated, the team may lack a DBA with deployment skills, but in other cases it seems to be simply laziness or lack of attention to detail. Not all databases automatically adjust to increases in CPU and/or RAM (memory) allocated to the server or virtual machine. Likewise, certain workload types very often require different configurations out of the box. Moreover, the scale or size of the database and the number of concurrent users for the benchmark also require different setups. Unless you are

doing a simple educational or relatively small benchmarking effort, then you need to properly configure the database. The key point is you should do the exact same optimizations for a benchmarking effort as you do for production systems.

The next major area of database preparation that quite often gets overlooked is storage allocations. Most database benchmarking tools do not understand your storage logical units (LUNs) or their RAID characteristics such as stripe width and depth. Most benchmarking tools simply create the required benchmark objects (tables and indexes) in the default storage area for the user running the creation and loading step. Some of these tools do offer an option to create partitioned objects, but once again with very little smarts as to your storage setup. What I tell people is to run any database benchmark as two separate major steps: create/load vs. execute. The idea being to run the first step in isolation so that you can modify or re-create the database objects to best utilize our storage layout. Then take a backup of that properly laid-out data such that for future iterations of the same execution you can just do a restore. I've even seen one database benchmarking tool that's smart enough to offer support for such a methodology (Quest's Benchmark Factory). Trust me when I tell you that a 3 TB TPC-H will run better and quicker if you only ask the benchmarking tool to load the data the first time. I've seen a 3 TB TPC-H take three days for the initial load, but restoring the backup only took three hours. When you are on a two- or three-week schedule, that one difference can make or break your efforts.

The last area of database preparation that also gets ignored is proper statistics and histogram collections, as well as running pre and post execution performance metric snapshots for test iteration comparison reporting. None of these tools can replace your DBAs' experience and insights. Some tools offer pre and post script allowances such that you can hand code those items and manually add them to the project. Very few offer that built into their workflow or graphical user interface. So your DBA will need to learn your database benchmarking tool, what it does vs. what

it doesn't, and then add their expert insertions where appropriate. I've seen oversight of this issue resulting in poor ability to diagnose or optimize performance. When done right, I have seen it make as much as a 400% difference. It's well worth paying attention to this.

Benchmarking Time Line

How long should a database benchmarking project take? It should be based off the mission statement and the answers to the dozens of questions it raised, and hopefully got answered. Then you should form a project plan that includes all the prerequisites necessary to even begin. Yet I show up at many efforts and the approved plan is we have two weeks to run a TPC-C or TPC-H that shows acceptable TPS. In one case, running a TPC-H, the equipment had not been deployed, the software had not been installed, and resources were tight to get such work done. Then they tell me we have to run several TPC-H runs at 3 TB or larger. Imagine their surprise when I said to just report today that effort failed and save the time and effort. Of course, they thought I was joking, so we began working on the project. It took four days to get the hardware and software ready to go. It took three days for the initial data load. So on Monday of the second week, we had were starting from ground zero with just five days left. This is not the exception; almost every database benchmarking project seems to be so misestimated as to severely limit any legitimate chances for success.

In the ideal scenario where all the hardware and software are set up, the correct server or virtual machine size preselected, both the OS and database optimized, the proper resources available for executing and adjusting the test runs for adjusting for iterative testing, and with a clear goal in mind, then two weeks might (and I only say might) suffice. I tell people who ask to figure on three to four weeks to start as an initial estimate, at which point most say we don't have that much time. When I question that limitation, I am usually told we simply have to start and see

where we get in two weeks – and then stop. I will ask those who won't just fire me on the spot, would they like a surgeon to start a procedure and then just quit at two hours no matter where they were in the procedure? Of course the answer is always no, but unlike a doctor, most of these professionals are hesitant to go back to management and say more time or forget it. So in those cases we try our best in the two weeks, but more often than not we do not achieve the desired results.

Another key aspect of the timing is what happens when the incorrect or insufficient resources are allocated? Remember that 3 TB TPC-H project that I mentioned above? Well the hardware that had been ordered for the storage was one direct connect storage enclosure with 15 spinning disks. I explained that the nature of the TPC-H benchmark was such that it was directly correlated to the number of spinning disks, and that all published benchmarks at that time of the same size were done on no fewer than 100 disks. Of course we were not going to get 85 more disks, as they could only afford to go to a second enclosure and a total of 30 disks. That took a week to get in place and ready to go. So see how these projects almost never finish in just two weeks? Do not let yourself be thrown into the deep end of the pool without a life preserver. If you are asked to perform a database benchmarking project in a too limited time, then just say no.

OK – I know it's damn near impossible and reckless career wise to tell management "no" (even though I've done it lots of times, maybe that's why I have never made it big time). But do try to fight for more time or to lower the expectations or to lessen the scope. Had that 3 TB TPC-H project been open to the idea of reducing the database size to 300 GB (the lowest meaningful size for that database benchmark) and getting the second disk enclosure, we could have succeeded. We could have set up and run the tests to get all our ducks in a row the first week, and then the second week added the second enclosure and rerun our tests to get the best final numbers. So look for creative solutions.

Benchmark Sizing

I am always surprised when people ask me sizing questions such as how big should my TPC-C database be and how many concurrent users should we run? As the outsider coming on to help, I really don't know their existing systems or the size of their end-user pool. You are going to be far better equipped than any outside consultant, but they can assist. So you should drive that decision.

A while back I helped a large national insurance company with their benchmarking efforts. They were considering a switch from mainframe to client server and relational databases. They were OK with running an OLTP-based benchmark as a gauge. In the beginning they were only running 100 concurrent user sessions for the benchmark. The reason was the 100-user limit on a freeware tool being used. As this was a very large company, that was simply not acceptable. The management quickly bought a commercial license, but the DBAs initially chose just to run 2,000 users. When I inquired (and then really pressed) as to why, the answer was that running the benchmarking tool was easy for up to that number, but required far more work to scale higher. So I asked since I know this system must handle your entire national network of agents, what is the maximum number of concurrent sessions you see today? The answer was 15,000 to 20,000. So then we had to buy additional licenses of the database benchmarking tool in order to deploy it as a console with 10 agents on other servers where each agent ran 2,000 concurrent sessions for a total of 20,000 concurrent sessions. Now knowing that the TPC-C spec says 1 warehouse per 10 users, we knew our scale factor had to be set to 2,000. which translated into a database size of 150 GB.

Now that all sounds well and fine in the case of OLTP, but what if you instead have to run a data warehousing (TPC-H) or decision support (TPC-DS) type benchmark? First and foremost, try to figure out in advance just how much data senior management wants to keep on hot, quickly accessible storage. I've heard businesspeople say we want to keep data

forever. But the hot data often is really just a manageable subset. For example, maybe the retail grocery chain wants to perform basket or trend analysis on what their customers buy during a typical year, also considering weather conditions, holidays, and sporting events. With today's fairly cheap storage costs, the answer might be as high as five to seven years' worth of detailed transaction data, which easily total tens or even hundreds of terabytes. Plus you might also be required to keep aggregate or summary data as well, plus indexes and any other structures that consume space. So benchmarking for these needs with say between 3 and 30 TB might make sense. The other factor you need to nail down is how long will executives in this environment wait for a meaningful strategic report. Remember, too, these data warehousing and decision support type of queries tend to process tens or even hundreds of millions of rows to form answers. High-level executives may be willing to wait four hours for an answer, which can directly and positively affect the bottom line. So you are going to need sufficient data to approximate these types of environments.

The single most important issue to keep in the forefront of your mind is that disk space is cheap and thus the business is wanting ever-bigger data sets. It won't do anyone any good to size your benchmarking database too small. Just remember that the database benchmarking tools are very slow on the data load process so you'll want to use the previously mentioned split project and backup method for expediency.

Workload Capture

Sometimes we're lucky enough to have the tools and the ability to capture real workloads and then replay them as our database benchmarking approach. Some of the previously mentioned tools offer such capabilities, as well as a few of the database vendors. The critical question becomes when is a good time period to capture such a workload? Once again it

will be far better to question your own in-house people who know the application than some outside consultant. You may well find that you need to capture multiple workloads: a typical business day, a typical business night, plus one or more business event periods. The first obvious issue is selecting the proper number and timings of workload periods to capture. But another issue often forgotten until the last moment is making arrangements for the extra overhead on the production servers for the capture processes plus the extra storage to hold that metadata. Some production boxes are so overloaded as it is, that getting permission to put an extra 3–5% load on the system is a nonstarter. The other major issue with workload capture and replay to prepare for is that you must make a complete copy of the production database in order to be able to run the captured workload. So more overhead in terms of the disk space for that copy plus the time to take a full cold or offline backup. Therefore, while workload capture and replay may seem like two very best ways to benchmark, the realities are that in many cases it's just not feasible. And that is without considering the extra cost for the workload capture and replay software, which for many platforms is quite expensive. You may very much want to do workload capture and replay but have to fall back to simple benchmarking. That's OK if your mission statement and answers to all of its questions can be met via database benchmarking instead of workload capture and replay.

Distorting Factors

This may at first seem like an odd section under the topic of preparation. But there are many factors to consider up front during preparation, which can have a major skewing or distorting effect on your database benchmarking results. If you're going to test on shared storage such as a SAN or NAS, can you at least have your own LUNs? Are there periods on a shared SAN or NAS where the normal maintenance tasks such as backups

can be avoided? Can the database server and benchmarking machines be kept on a segregated network segment? There are simply too many peripheral issues that could sneak up and bite you. Let me give just a few examples of things that were not accounted for prior to benchmarking and which totally invalidated our results.

On one occasion a network switch firmware upgrade was done one weekend during our benchmarking efforts. We had not been aware of it, and everyone just assumed it could in no way harm our efforts. Wrong! Somehow of VNETs and Jumbo Frames settings got hosed during the firmware upgrade. We spent two weeks spinning our wheels before we could figure out and prove that some external factor had skewed our results. It's not supposed to happen and yet it did. And this kind of issue was a bear to troubleshoot. So when you get radically different benchmarking results one week vs. another, then do look for any rare causes. Because generally speaking, database benchmarking results really should not vary by more than 5–10% when done right.

On another occasion while testing an Oracle Real Application Cluster (RAC) setup, our results one week again just went haywire. We looked for nearly four days before someone spotted that one of our bonded network cards was throwing low-level errors not bubbling up to the OS logs. And because it was bonded it did not fault, it just ran much slower, which caused lots of issues for the cluster including nodes dropping for no readily apparent reason. At one point one of the DBAs wanted to log the problem as an official bug with Oracle that RAC could not scale to 20 nodes reliably. Don't assume that odd behavior can be ascribed to OS or database bugs. Sure, it could happen. But it will be the rare exception and not the rule. But it's very easy to get caught in that trap.

This last example is yet another strange hardware example. And yes, I know that I said these kinds of issues should be quite rare, but if you do enough benchmarks then you will encounter more rarities. We had a disk drive starting to fail that was part of a RAID 5 LUN. The disk failures were not quite critical yet and the RAID was compensating. But adding

those additional parity checks on almost every IO added up to the point of skewing the results. We were very lucky as we caught this problem with only four hours of lost time in total. But when you're in the final week of a benchmarking project, getting results that suggest the other results are tainted is highly unnerving.

Summary

Although this was a fairly short chapter, it nonetheless covered one of the most important aspects of successful database benchmarking: preparation. You can never be too prepared. You should allocate sufficient time and resources. You should have a plan and know what your expectations are. Database benchmarking projects are too important and usually on a tight time line, so you cannot afford to fly by the seat of your pants. My advice would be to approach a problem like president Abraham Lincoln who said: "Give me six hours to chop down a tree and I will spend the first four sharpening the axe." That's preparation.

CHAPTER 5

Benchmarking Mistakes

As the prior chapter stressed, sufficient preparation or lack thereof can make or break a database benchmarking effort. For example, not just anyone has all the requisite background and skills to successfully run a database benchmark. They must know a lot about databases, especially proper configurations, performance monitoring, and tuning techniques. Likewise you need to have scheduled ample time and other resources for the effort. But assuming you've done all that, there are still many things that can go wrong during a database benchmarking effort. This chapter will highlight some of the major and most common missteps that plague many database benchmarking projects once they are underway.

The key takeaway from this chapter should be that you can do all the preparation and yet things can still go wrong. Remember the old saying: "The best-laid plans of mice and men often go awry." I can honestly say that none of the database benchmarking projects I've worked on went 100% as planned with zero speed bumps along the way. However, by knowing that luck, I simply suggest that you remain flexible and avoid intransigent behavior during the effort. Moreover, I suggest remembering what Robert Schuller said, "Problems are not stop signs; they are guidelines." The central idea presented in Chapter 1 is that database benchmarking by definition endeavors "*to make the database sweat.*"

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_5

Therefore, you should reasonably expect to hit boundary conditions or you're not pushing the system hard enough.

Keep References Handy

In Chapter 2, I stressed how important it is to read the database benchmarking spec (if one exists) before trying to run some software that implements that benchmark. The reason was to understand what it's all about and to be able to answer any questions the software might ask, which might expect your answers to be within spec guidelines. For example, the TPC-C says 10 concurrent users per warehouse; therefore if you want to test with 10,000 users you need to set a scale factor of 1,000. However, keeping the spec handy might be critical when you encounter those speed bumps that I mentioned always occur. Sometimes a potential database tuning technique you want to try may or may not be allowed to overcome those issues. The three most common questions include:

- Is adding additional indexes or modifying the current indexes permitted (e.g., switch column order)?
- Are table and index partitioning allowed, and if so what are the rules governing their implementation?
- Are collecting statistics and histograms allowed, and if so what are the rules governing their implementation?

The benchmark spec will generally answer if these actions are allowed or not, but they do not advise you on the detailed mechanics of doing so. Thus while horizontal partitioning might be permissible, there is absolutely no mention of what partitioning type (e.g., range vs. hash vs. other), what partitioning key, or what number of partitions for any given size database. Furthermore, neither the spec nor your benchmarking tool knows about the particular hardware you have to work with, nor what particularly your test hopes to accomplish. Only you know those answers,

and so you're going to have to be prepared to answer such questions. For that kind of information, another handy reference document is the published test results from the tpc.org website. The document to locate is the full disclosure report from whomever has run the same benchmark on similar hardware for the same database and size that you are using. There you will find detailed database configuration options and database object creation scripts. This info previously was contained in Appendix B (database design) of published full disclosure reports, but now that info is in the supporting files – typically file #2. This is the best place to find examples of just what partitioning and indexing schemes others have found to be optimal.

Note The tpc.org website currently seems to have far less published full disclosure reports than in days past. In fact, some database vendor results now appear totally absent.

All the Required Tools

In Chapter 1, I likened database benchmarking to a cardiac stress test. That's the procedure where a doctor, most often a cardiologist, hooks you up to a heart monitor and has you walk briskly or run on a treadmill for some prolonged duration. The doctor's goal is to "*get your heart rate up*" and "*to make you sweat.*" I like to reference Figure 5-1 shown here whenever discussing what's required to successfully perform a database benchmark.

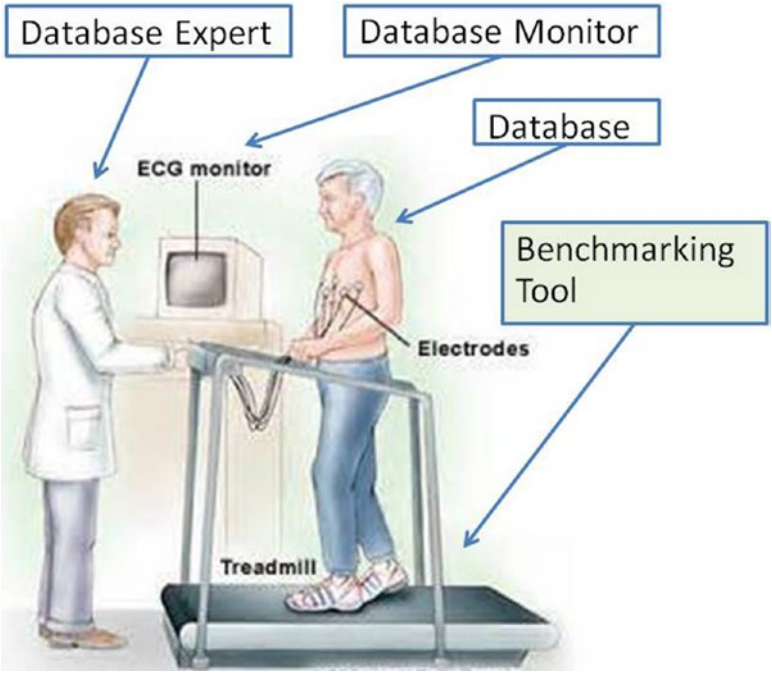


Figure 5-1. *Benchmarking like Cardiac Stress Test*

For a cardiac stress test, there is a doctor, the electrocardiogram (ECG) monitor, the treadmill, and the patient. Likewise the database benchmark has a database expert, often a DBA, a database monitoring tool, their database, and the database benchmarking software. For the cardiac stress test the doctor will decide what treadmill inclination, what speed, what duration, what to monitor, how to diagnose, how to treat, and so on. Similarly the database expert will choose a benchmark, it's database size, the number of users, the duration, and so on. Seems simple enough – right?

The one recurring, major problem that I encounter when assisting with database benchmarking projects is that the database expert (or the nonexpert if none is assigned) mistakenly assumes that the benchmarking tool will monitor the database, report on the issues, and then make recommendations. I have to remind them that the benchmarking tool is

just the treadmill. Its only job is to stress the database to make it sweat. How or why they expected the database benchmarking tool to do it all is a mystery to me. I guess it's just wishful thinking, but I see it in about 25% of the projects I help. It's often the first problem we have to address when I join the project. Finally remember that most database monitoring tools only show the symptoms, they do not give advice on the corrective actions. Plus even if they do, often that advice is rather generic in nature and not specifically attuned to your hardware and workload.

No Big Red Easy Button

This last issue, expecting the database benchmarking tool, is actually just one symptom of a debilitating condition I commonly refer to as "*Looking for the big red easy button.*" So I show people Figure 5-2 in an effort to explain that while the office supply store Staples may market that they make all ordering very simple, showcasing the big red easy button, there is in fact no such thing when performing database benchmarking. That no benchmarking tool is an all-in-one stop offering a big read easy button experience. Astoundingly this seems to shock most people. Database benchmarking is hard work no matter how you slice it. There is simply no getting around this fact.

Figure 5-2. *The Staples Office Store Big Red Easy Button*

Most database benchmarking tools work fairly similarly, and thus most database benchmarking efforts proceed in much the same way. As such, let's examine what basics steps any successful database benchmark effort, regardless of database benchmarking tool, must perform in their entirety in order to meet expected results. Understanding these steps will help to better explain what's meant by "*no big red easy button!*"

1. Configure and optimize an isolated network segment.
2. Configure and optimize the server hardware.
3. Install and optimize the server operating system.
4. Install and optimize the database software (e.g., Oracle, SQL Server, PostgreSQL, MySQL, etc.).

5. Create and optimize the "TEST" database.
6. Create user or schema to own the benchmark's database objects (i.e., tables and indexes).
7. **Create and load the benchmark database objects to their initial, pre-workload state (obviously dropping any preexisting database objects).**
8. **Run the benchmark workload or transactions as defined by the spec (in most cases altering the data, thereby requiring reload if run again).**
9. Monitor database performance under workload, diagnose, and then optimize.
10. Repeat steps 7, 8, 9, and 10 until satisfied (i.e., results approximate or match expectations).

In this oversimplified 10-step process, at best benchmarking tools can really only automate steps 7 and 8 (highlighted in bold above). Also note that those steps are inherently inefficient in that execution iterations, which is a best practice and quite common, requires the database benchmarking tool to reload all the data each time. In the case of a 30TB TPC-H, that might take a week! So there must be a better, more efficient way. I suggest replacing steps 6,7, and 8 with the breakdown below.

- Create two database schemas:
  - SCHEMA-1 for holding a static copy of the "*default*" database objects as created by the benchmarking tool without any permissible structural optimizations
  - SCHEMA-2 for holding a structurally optimized copy of the database objects for an execution iteration

- Create the database benchmarking tool's standard project definition for the desired benchmark of a given size.
- Separate the standard benchmarking tool's project into two distinct phases:
  - PHASE-1 to create the database objects and load the data into SCHEMA-1
  - PHASE-2 copy all the data from SCHEMA-1 to SCHEMA-2 and run the next iteration's execution
- Run the PHASE-1.
- Back up or export SCHEMA-1 (for any unplanned failures).
- Manually write a SQL script to create optimized copies of the database objects, and load them using "create table as select" (CTAS) (runs on database server so very quickly).
- Manually add the CTAS SQL script to PHASE-2 copying from SCHEMA-1 to SCHEMA-2 and then running the test.
- Run PHASE-2 to execute a test as many times as you like.

While this takes the simple 10-step process to 15 steps, it actually makes the process easier to understand (i.e., divide and conquer) and makes repetitive test execution several orders of magnitude faster. Plus now if you want to add other SQL scripts to collect statistics and histograms, you now have proper segregation of processing and locations where you can add them. So as you can see, the idea of a big red easy button or simple benchmarking is patently false. Database benchmarking is tough work and requires attention to detail plus manual work to optimize an otherwise inefficient process.

Proper Tool Deployment

While most of the database benchmarking tools are easy to download, install, and run, they also often have far more complex deployment methods that few users research or utilize. Failure to understand and leverage such deployment models can radically skew results or even result in failed benchmark testing overall. Suppose you want to run a TPC-C with 10,000 concurrent users. Do you really think your lone desktop or laptop PC can send that much traffic to your database server? Even if you say yes, what about network bandwidth? There's only so much traffic a single gigabit Ethernet card can process. While the test might run to conclusion without error, does that really represent what the database server could have really achieved? Yet many people do just this and wonder why the results are not repeatable nor as good as they had expected.

Let's examine a typical client/server type deployment model that many database benchmarking tools provide to accommodate running such workloads. For this demonstration I'll use Quest Software's Benchmark Factory (BMF). Instead of just installing and running BMF on a lone computer, you can deploy BMF agents on other servers to construct a distributed architecture such as the one shown in Figure 5-3.

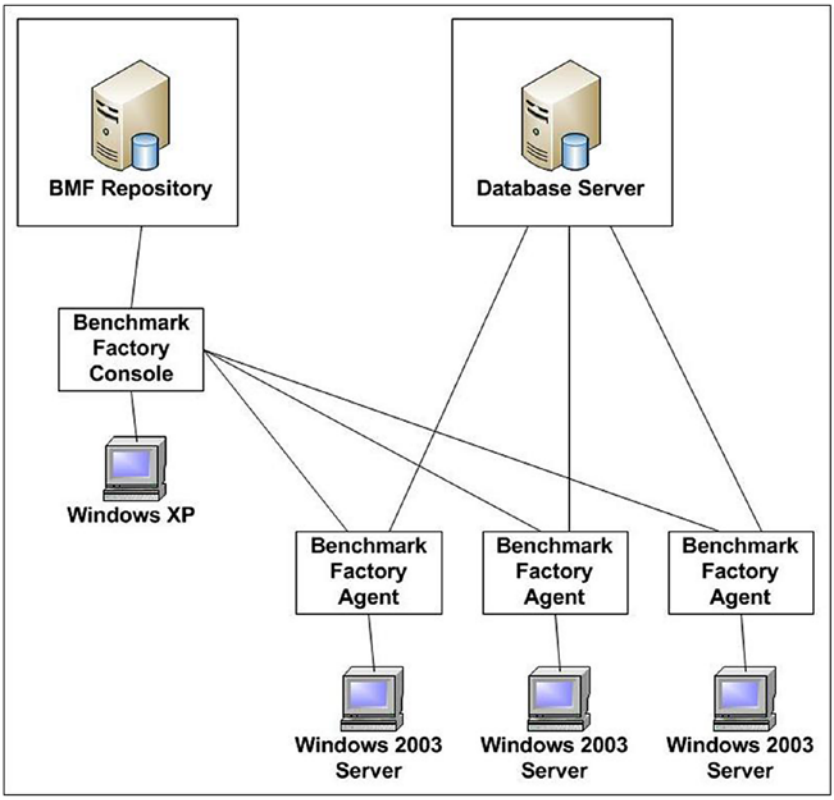


Figure 5-3. *Benchmark Factory Distributed Architecture*

The central concept is that the desktop or laptop PC simply runs the central control console, not the actual database user sessions requesting work from the database server. There are two main advantages to this approach of distributing the agents. First, the user sessions and their workloads can be spread across many servers, and therefore across many CPU's and memory. Second, the agent servers can be in the server room with high-speed network connections to the database server such that network bandwidth and latency don't skew the performance results. I've worked on benchmarking projects where we had as many as 120 agents for a given benchmark test run. This is really the only reliable way to run such

large tests. Note that while some tools such as BMF provide an automatic mechanism to automatically deploy the agents from the console, other tools require installing the software and then manually linking them up by their network IP address. That might provide sufficient reason to justify the extra cost of a commercial tool like BMF.

Unexpected Human Factors

I am going to keep this section very brief since I don't want to sound like an evangelist on a soapbox. But nonetheless you need to at least be aware of the possibility that factors other than simple database performance may be the true agenda for benchmarking. Knowing this agenda can sometimes steer the effort in a direction that best suits reaching the necessary results.

If those requesting or performing the database benchmark have already made up their minds, then the benchmarking itself effort may well just be a "*checklist*" item to cross off on the way to some presupposed conclusion. This is quite simply human nature. It's neither wrong nor avoidable. Being aware of these kinds of human factors can help to achieve real success.

Validate Our Direction

Suppose your management says, "We're moving to the cloud, so you need to provide benchmark results showing that the performance is comparable or better to complement the cost savings argument." So why test? All the cloud vendors offer numerous virtual machine configurations (e.g., general purpose, memory intensive, CPU intensive, IO intensive, etc.), plus offer scalable CPU and memory resource allocations for each configuration type. Possibly what management is really asking for is what's the right-sized cloud virtual machine image that can deliver all the required, key performance metrics. That radically changes the scope

of the benchmarking project, because now you need to both provide a recommendation and its performance. But this type of clarification often does not become clear until the benchmarking project is underway.

Self-Fulfilling Prophecy

Now instead suppose your management says, "We're replacing our Oracle SPARC/Solaris servers with Intel/Linux servers, so you need to provide benchmark results showing that the performance is comparable or better to complement the cost savings argument." But there's a major wrinkle/issue embedded in this simple statement. What if many of the DBA's for current Oracle databases on SPARC/Solaris who end up performing this database benchmarking are not happy about the move? Don't laugh, it happened on a major benchmarking project that I was involved with. There simply was no convincing those DBA's that the new Intel/Linux servers could ever measure up. I won't say that sabotage occurred, but I can say what should have been an easy effort was not and took far longer than planned. Once again, you may well not spot such issues until the benchmarking project is underway.

Top Ten Benchmarking Misconceptions

I have great respect for the science of database benchmarking and all those who practice it. Yet it's not a science that lends itself to novice or ignorant attempts without proper preparation and all the necessary tools. However, whenever I join a benchmarking effort already underway and who has encountered issues, I find some common missteps. Below are the top 10misconceptions that I encounter on a regular basis (some of which have been mentioned in earlier chapters).

#1: I'm using a tool like Quest's Benchmark Factory, so that's all I need

Wrong. I highly recommend that anyone doing benchmarking read the specs for whatever industry standard tests they are going to perform. This is because software to automate these tests will ask questions or present options that you cannot really define unless you understand their context, which is defined in the spec.

For example, the highly popular "OLTP" test known as the "TPC-C Benchmark" (http://tpc.org/tpcc/spec/tpcc_current.pdf) defines "scale" factor as follows:

> ***Section 4.2.1:*** *The WAREHOUSE table is used as the base unit of scaling. The cardinality of all other tables (except for ITEM) is a function of the number of configured warehouses (i.e., cardinality of the WAREHOUSE table). This number, in turn, determines the load applied to the system under test which results in a reported throughput (see Clause 5.4).*
>
> ***Section 4.2.2:*** *For each active warehouse in the database, the SUT must accept requests for transactions from a population of 10 terminals.*

So when a tool like Benchmark Factory asks for the scale factor, it does NOT mean the number of concurrent users – but rather the number of warehouses. So a scaling factor of 300 means 300 warehouses, and therefore up to 3,000 concurrent users.

This requirement to read the spec is critical. It will be an underlying issue for every remaining misconception and problem that I'll mention below.

#2: I have an expensive SAN, so I don't need to configure anything special for IO

Wrong. The size, type, and nature of the test may require radically different hardware settings, even all the way down to the deepest level of your SAN. For example, a data warehousing test like the TPC-H is best handled by a SAN whose "read-ahead" and "data-cache" settings are set more for read than write, while the OLTP TPC-C would benefit from just the opposite. Relying on defaults can be a really big mistake.

Likewise, the SAN hardware settings for stripe depth and stripe width should be set differently for these different usages. Plus of course the file system and database IO sizes should be a multiple of the stripe depth. In fact, a common rule of thumb is:

```
Stripe Depth >= db_block_size X db_file_
multiblock_read_count
```

Furthermore, selecting the optimal hardware RAID level quite often should factor in the benchmark nature as well. Where OLTP might choose RAID-5, data warehousing might be better served by RAID-0+1.

Finally, the number of disks can also be critical. For example, TPC-H tests start at around 300 GB in size. So anything less than 100 spindles at that size is generally a waste of time. And as you scale larger, 800 or more drives becomes common as the minimum recommended setup. The point is that no SAN cache is ever large enough for monstrous data warehousing queries' workload.

I've seen **up to 500% result differences** when varying SAN settings and the number of disks.

#3: I can just use the default operating system configuration right out of the box

Wrong. Most databases require some prerequisite operating system tweaks, plus most benchmarks can benefit from a few additional adjustments. For example, I've seen from 50–150% benchmark differences running TPC-C benchmarks for both Oracle and SQL Server by adjusting but one simple file system parameter. Yet that parameter is not part of either database's install or configuration recommendations.

Now you might argue that you can skip this step since it will be an "apples to apples" comparison because the machine setup will be the same across tests. OK, but why potentially wait three times as long for worse results. Since a 300 GB TPC-H test can take days just to load, efficiency is often critical in order to meet your time deadlines.

#4: I can just use the default database setup/ configuration right out of the box

Wrong. While some databases like SQL Server might be "universally useful" as configured out of the box, other databases like Oracle are not. For example, the default number of concurrent sessions for Oracle is 50. So if you try to run a TPC-C test with more than 50 users, you're already in trouble.

Likewise the nature of the benchmark once again dictates how many various database configuration parameters should be set. A TPC-C test on Oracle, for example, might well benefit from init.ora parameter settings of:

```
SGA_TARGET & SGA_MAX_SIZE = 60% memory

PGA_AGGREGATE_TARGET = 20% memory

CURSOR_SHARING = SIMILAR

OPTIMIZER_INDEX_CACHING = 80

OPTIMIZER_INDEX_COST_ADJ = 20
```

I've seen **as much as 533% database benchmarking performance improvement** from just adjusting a few such parameters, so imagine what a careful review of all the database configuration parameters for the test nature could provide!

#5: Tools like Benchmark Factory will create optimally designed database objects for my hardware

Wrong. Software like Benchmark Factory is simply a tool to automate the complex and tedious process necessary to execute a benchmark. For example, the TPC-H is simply a collection of 22 very long and complex SELECT statements against a very simple database design with some really big tables. It tests the database optimizer's efficiency in handling complex statement explain plans and their executions. So optimal database structural design could play a major role in the performance achieved.

Yet the simple CREATE TABLE statements from most database benchmarking tools is suboptimal. I refer you back to this chapter's earlier section about the big red easy button and the 15 recommended steps for performing a database benchmark. You should manually optimize your database object structural design for the best results, even if the tool like Benchmark Factory offers some limited capabilities to do so since the tool cannot know your hardware.

#6: Tools like Benchmark Factory will automatically monitor, tune, and optimize all my hardware, operating system, and database configuration parameters

Wrong. As was stated in the prior issue, database benchmarking tools are simply the treadmill designed to make a database sweat – and that's it.

A tool like HammerDB runs queries and/or DML to make the database work hard for a specified scenario in order to learn where any bottlenecks might exist – but not to find or identify those issues.

If you want to monitor, diagnose, or tune/optimize your database for such tests, you'll need the appropriate tools like Quest Software's Spotlight for Oracle or Oracle Enterprise Manager with the Tuning and Diagnostics packs. Remember, database benchmarking tools like Benchmark Factory are simply a *"load generator."*

#7: I don't need a DBA to perform benchmark tests – anyone technical can do it

Wrong. Look at all the issues above again – sometimes database developers or other simply technical database savvy people may not be cognizant or authorized to make such decisions. The key point once again is that benchmarking requires more than just someone to run tests – it requires someone who knows benchmarking and can speak to all the issues above. Otherwise, you'll simply get results that are not really what your hardware could do. And 500+% performance differences mentioned above are more than just worthless background noise; you could make a mistaken strategic decision with such misinformation.

#8: I can very easily compare database vendors on the same hardware platform

Possibly. If you have one or more DBA's who can address all the above issues for each different database platform, then by all means yes. Otherwise, you simply cannot compare the databases reliably by simply installing and running the same test for each. There are far too many dependencies and variables to trust such a simplistic approach.

#9: Transactions per Second (i.e., TPS) are what matter most in benchmarking

Rarely. I think TPS is one of the most misleading values – but everyone seems to focus on that one first. Let me explain. As you increase the number of users or tasks being performed, by definition you're actually increasing the TPS. At some point it may plateau, but what does that tell you? Figure 5-4 below shows the results from running the TPC-C benchmark while changing various Oracle configuration parameters from misconception #4 above. We don't see the 533% improvement that was quoted by looking just at the TPS numbers.

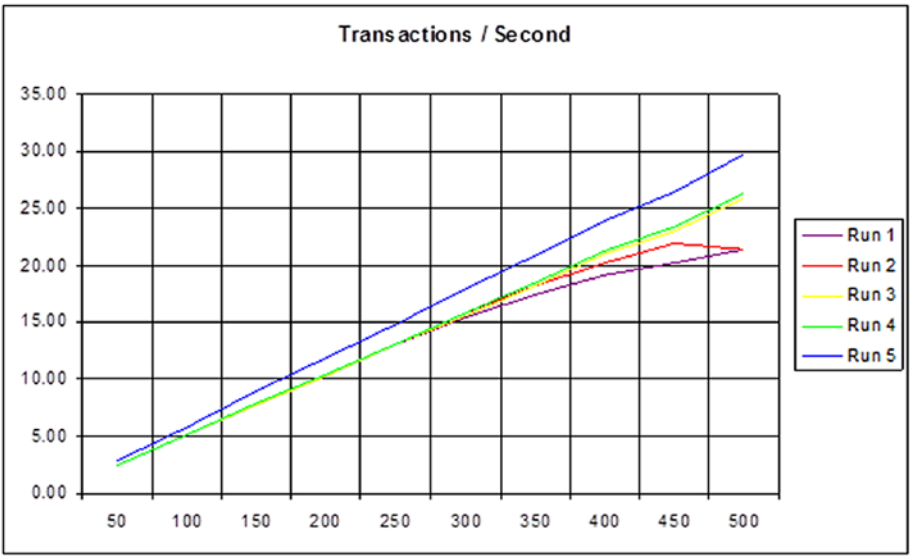


Figure 5-4. *TPC-C Results for Transactions per Second*

However, when we look at the average response times for those same test runs as shown below in Figure 5-5, now the 533% number should be far more evident. Test run #1 with default database configuration settings had an average response time of 5 seconds at 500 concurrent users, while

test runs #4 and #5 were around a fifth of a second average response time. Not only does average response time mean something real in user or SLA terms, but it's far more obvious and understandable.

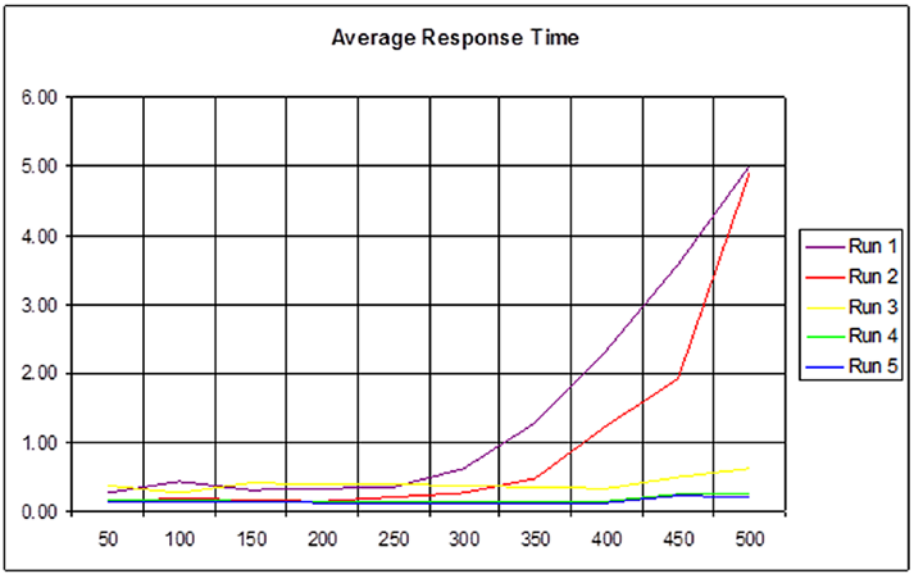


Figure 5-5. *TPC-C Results for Avg Response Time*

#10: We can spend just a few days and benchmark everything we're interested in

Never. Almost universally, the proper setup of all the dependent components can take a week or two – and when doing large-scale benchmarks, you can sometimes add up to a week for data loads and index creation. So make sure to carefully project your time to complete such efforts. Because there will be surprises along the way, even if you do follow all the above advice.

Summary

This chapter covers most of the common surprises and gotchas that you will invariably encounter when performing database benchmarking, even if you have followed all the recommended preparation in prior chapters. What it boils down to is that benchmarking is far too complex for anyone to identify every possible variable or issue likely to skew the effort and its results. Thus, being ready to handle these on-the-fly issues is critical. The chapter wraps up by detailing the top 10 misconceptions seen on many database benchmarking projects.

CHAPTER 6

Benchmarking Hardware Options

The first five chapters have educated and prepared you for performing database benchmarking. Now it's time to leverage that knowledge. This chapter focuses on issue if you're working on physical servers or "*raw iron,*" and not working with virtualization or the cloud. While both of those paradigms are both new and hot, nonetheless there still remains the need on occasion to work with on-premise database servers. Your company may not be quite ready for database deployments on a hypervisor or the cloud, plus you may need to benchmark on premises as a baseline in order to compare to virtualization or the cloud. There are many other reasons as well, so let's now look at what one should do when benchmarking on premise.

Note While this chapter may highlight examples from a given hardware setup, the principles and concepts shown throughout this chapter would remain the same. So don't focus on any particular hardware or vendor mentioned.

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_6

Draw a Picture

When working with on-premise hardware, the first and often critical step is to construct an architectural diagram of all the components from top to bottom, including all networking and storage. There are several reasons for doing this.

- Serves as a common reference for comprehension, planning, and rollout of all the necessary hardware
- Provides a complete road map of all the moving parts required for problem diagnosis and resolution
- Should be included with an deliverables or published documentation describing the benchmarking effort

While all three reasons have merit, it's the road map for problem diagnosis that is often most critical. There are simply far too many required components whose sum efforts contribute to the experienced or perceived performance of any database application. That's vital because the database might not always be the underlying reason for a performance issue. As I've mentioned in earlier chapters, sometimes a misconfigured network switch, a bad network interface card (NIC), or disk storage LUN configuration may be the real culprit that just makes the database appear slow. So having this info could make or break a database benchmarking effort. Look now at Figure 6-1, a 10-node Oracle Real Application Cluster (RAC) that I once benchmarked.

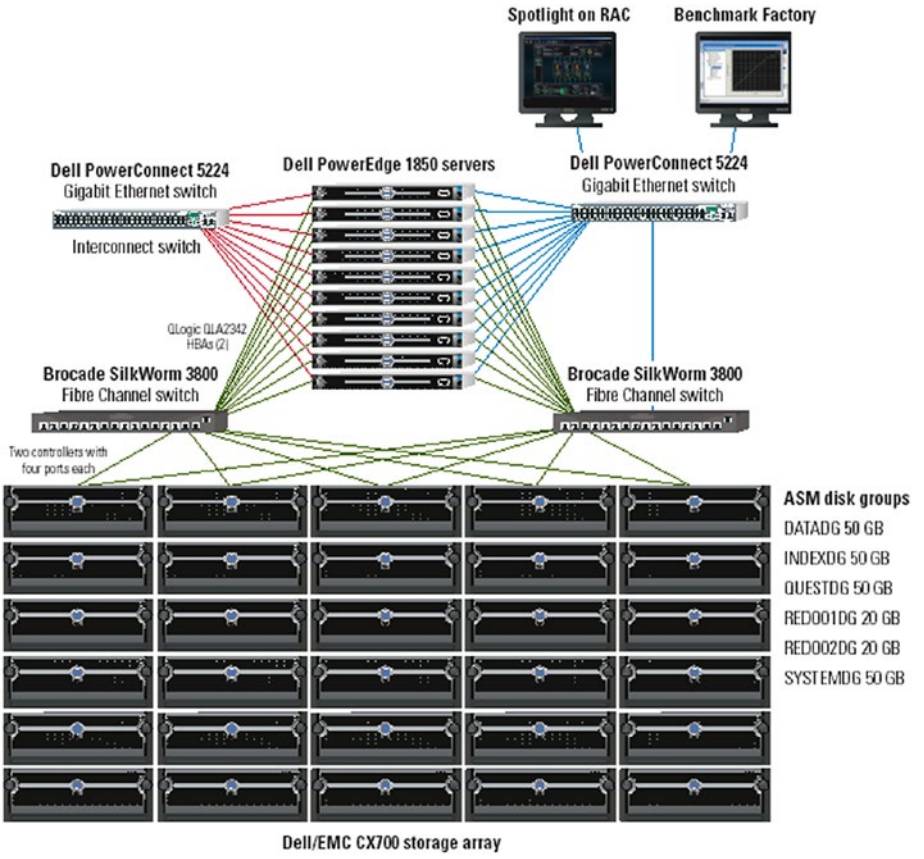


Figure 6-1. *Ten-Node Oracle RAC Cluster*

Remember the cardiac stress test analogy, so here we have the Oracle database on the Benchmark Factory treadmill connected to a cardiac monitor called Spotlight on RAC. However as detailed as this diagram may appear, it is lacking the actual storage information for RAID level, stripe width and depth, disks per LUN, and SAN or NAS controller/caching information. So it may well be that you'll need a couple of diagrams to have a complete overview of all the critical hardware components and their configuration. On a typical effort it's not uncommon to need three to five diagrams to have a full understanding.

Sometimes the team responsible for performing a database benchmark will question the need for such diagrams. They have the servers all connected and ready to go. It's the database that's being benchmarked, and if any issues do arise they believe the picture is really not necessary. Usually I counter with the question: Could you drive from New York City to Dallas with no road map, no GPS, and no smartphone without getting lost or the trip taking longer than it should? Imagine the database monitor reports wait events such as very high latency or IO reads and writes. Many DBA's would focus first on the SQL statement most responsible for those waits. But what if the database design is sound, the query well written, and the execution plan is fairly optimal? You could spend time trying to get "*blood out of a turnip*" by assuming that the database needs some magic configuration parameter changed, a new index on the largest tables, or experimenting with the SQL in order to identify a better execution plan. Sometimes that is the right approach. But when you are on purpose stressing the system as a whole, it's also very likely the breakdown or failure of any component is to blame. You must keep an open mind and view of where the problems could be in order to get the highest possible database performance from that system.

CPU and Memory

Often the initial system components that are thought to be the most critical are the CPU and memory. For the CPU, what's the count of the sockets, how many cores per socket, and is hyperthreading available? Then for memory, how much RAM can be allocated to the database? Furthermore, how should that RAM be distributed among the various database memory allocations available? These obviously are all important issues. But when you purchase an automobile you wouldn't make the two most important issues the engine's horsepower and gas tank size. There are many other factors such as network latency and maximum IO rates that could well

be the true limiting factors. Yet it's human nature to focus on things easy to brag or boast about, such as the system has 64 threads. But it's not uncommon to see database benchmarking efforts utilizing less CPU and memory than the server possesses.

OK, now that I've said not to focus on CPU and memory, there is one exception. Some database versions or editions may limit the CPU and memory that the database can actually use. For example, Microsoft SQL Server 2017 has the CPU limits shown in Figure 6-2.

SQL Server edition	Maximum compute capacity for a single instance (SQL Server Database Engine)
Enterprise Edition: Core-based Licensing*	Operating system maximum
Developer	Operating system maximum
Standard	Limited to lesser of 4 sockets or 24 cores
Express	Limited to lesser of 1 socket or 4 cores

Figure 6-2. *SQL Server CPU Limits*

So imagine you're benchmarking a standard edition on a server with two sockets, where each CPU has 16 cores plus hyperthreading for a grand total of 64 threads. As this is way over the limit of 24, SQL Server will leverage only a portion of the total CPU capacity. In fact, since we want to make sure that SQL Server always uses the most efficient CPU at any given time, we may want to disable hyperthreading so that we are sure the CPU's that SQL Server does use are full blown, real CPUs. Thus it may be advisable to disable hyperthreading in the server's BIOS. Your mileage will vary, so it is best to test this and not just take it as fact.

Spinning Disks

For this section we're going to limit our discussion to traditional spinning magnetic media disks. The same concepts contained here will apply equally regardless of whether SAS or SATA disks.

The Achilles' heel of any database is IO. No matter how many CPUs the database is using, they each will have to wait on IO at some point. In fact, with CPUs having such high cores per socket and hyperthreading, it's fairly easy these days to have the CPUs outpace the disks. Moreover some database benchmarks like the TPC-H (and possibly TPC-DS) are generally performance constrained by the number of spindles. Furthermore, issues like the number of controllers or heads on a disk array, the amount of cache, and certain firmware settings all contribute to the overall IO bandwidth that is achievable. Then there's the RAID configuration that for databases can be critical. Let's look into these.

Let's assume that you are benchmarking on a non-shared or dedicated NAs or SAN, so that you can request changes pertinent to your benchmarking efforts. One often-overlooked aspect is the firmware's read-ahead algorithm choices and read vs. write cache sizing. I once had a TPC-H benchmarking effort where the user was not happy with the performance results we had achieved. We had not drilled down to this level of optimization details. We then changed the firmware settings and saw the performance bump that we needed to complete the project. Sometimes little things do matter.

Another area of benchmarking frustration (as well as being a very bad practice for production) that I encounter is with the overutilization of RAID 5 or 6 for databases. I understand that everyone wants to do RAID and we all think bigger is better (where bigger means more total usable space). But I always stress to people that for optimal database performance for database benchmarking (as well as production), the best level is RAID-10 (sometimes called RAID-1+0) regardless of the loss of space. It's better to

have just enough space that's fast rather than lots of space that's slow. Yet I always find myself reminding people of the RAID overhead shown in Figure 6-3.

I/O Impact		
RAID level	**Read**	**Write**
RAID 0	1	1
RAID 1 (and 10)	1	2
RAID 5	1	4
RAID 6	1	6

Figure 6-3. *IO Impacts of RAID Levels*

The number of writes on RAID-10 may be two, but that's two simultaneous writes of the same data to two different disks where often completion of either identical, concurrent IO signals success. Now examine RAID-5 and RAID-6 with values of 4 and 6 respectively. These writes are all different data. Some are slices of the actual data, others are parity bytes. The key point is that's a lot more total IOs that must complete, and they must all complete before the IO operation reports success. The actual performance degradation is not really 2X or 3X, but it is very, very noticeable. So avoid RAID-5 on all database benchmarking projects (and on production too).

We now arrive at one of the most often overlooked IO considerations when database benchmarking: stripe width and depth or size. Let's assume that your database block or page size is 4K. Let's also assume that your storage admin has created a RAID-10 LUN for you with 8 total disks, so there are 4 disks per copy of the data. Thus the stripe width is 4. Now a possible ideal stripe depth could be calculated as: N * block or page size, where N >=1. This is known as coarse-grained striping, which strives to maximize total overall I/O throughput by spreading many simultaneous IO requests across more disks. This is considered desirable for high

concurrency OLTP systems as well as data warehouses. Sometimes people instead (mistakenly) chose fine-grained striping where they prefer to have each single IO request spread across all the disks. They calculate their stripe depth as block or page size / stripe width. While this choice might make that single IO faster, it also makes handling many concurrent IO operations slower as they queue up. Remember that most databases need to handle lots of concurrent IO, so choosing this option can result in very suboptimal performance. Make sure you have a good reason whenever selecting it.

New Storage Technologies

Three great things have occurred the past decade with storage: it's cheaper, disks have grown very large (12TB/disk), and new storage technologies have emerged that are much faster than traditional spinning media disks. Let's examine some of the more common, newer technologies.

The first breakthrough were solid state disks or SSD's. These were essentially NAND chips packaged with a processor and controller to provide storage that looked and felt much liked traditional disks but operated much faster. Figure 6-4 shows an example of the architecture.

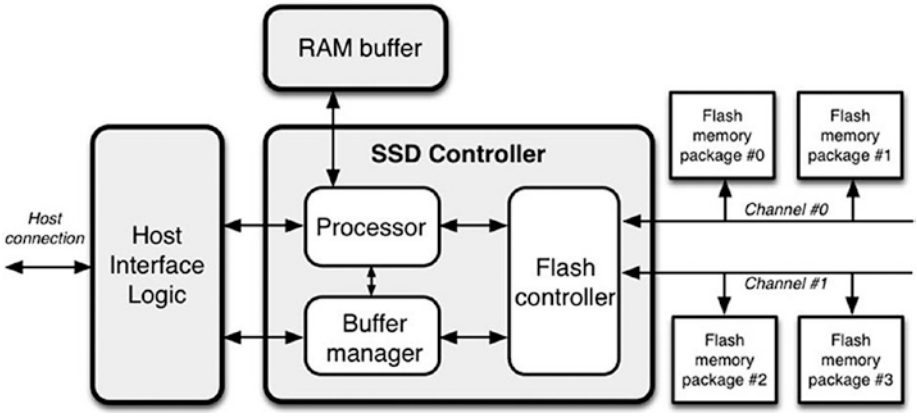


Figure 6-4. *Architecture of a Solid State Disk*

However, SSD's had one major drawback: they used the old AHCI and SAS/SATA protocols to access the NAND chips. These protocols were written and optimized for spinning media. As such they included logic for inner tracks vs. outer tracks and many other levels of no-longer necessary logic. Plus even though NAND chips could process data quickly, these protocols had a built-in bottleneck that they were hamstrung by the SAS/SATA data transfer limits of 600 and 1200 MB/s. Nonetheless SSD's ushered in an era of much faster IO.

The second generation of flash-based technology was basically SSD disk technology on a card on the fast PCIe bus, which provided a major boost of data transfer rates to between 2000 and 8000 MB/s. These cards could be placed on database servers, and in some cases on either SAN or NAS storage arrays. In fact, a whole new storage solution referred to as "*all flash arrays*" soon followed.

The third generation of flash-based technology was to replace the old AHCI and SAS/SATA protocols with non-volatile memory express or NVMe. This new protocol was written from the ground up to handle flash with no leftover baggage from legacy spinning disks. As such, NVMe offers data transfers rates of up to 32000 MB/s. That's over 53X faster than the first-generation SSD's.

Plus there are some exciting new technologies that will soon make mainstream, such as Optane, which offers near RAM memory-like speeds but with capacities like SSD's.

So how do these new technologies affect database benchmarking? The simple and obvious answer is you can place your databases on SSD or NVMe flash to get much better results. If your databases are too big for the flash-based storage, you can still place areas like TEMP, ROLLBACK, LOGGING, etc., on these devices to speed up many internal database operations. I've seen some data warehousing benchmarks where we had 30TB of data, so we could not put it all on flash, yet by using it just for temporary areas for sorting and grouping resulted in major performance improvements. Likewise, on OLTP benchmarks where flash was used

for logging and rollback activity alone, again the results were quite astounding. The point is even if you cannot put everything on flash, by selecting the right, limited set of items to place on flash, this can be beneficial.

Specialized Appliances

One last area where you may be asked to benchmark for is specialized database or IO appliances. These were quite popular before the major cloud movement. In essence these were simply hardware vendors constructing hyperconverged computer systems aimed at improving database performance, typically by both optimizing component selection and overall system configuration. Here are some examples:

- Oracle Exadata
- Oracle Database Appliance (ODA)
- Dell Fusion IO
- HP StorageWorks IO Accelerator
- Various Hyperconverged System Solutions

If you run into any one of these specialized environments where you are to perform database benchmarking, my advice is to hire a consultant familiar with such systems. For example, Oracle Exadata is very complex. I managed an Exadata system for four years. I learned so many new methods for optimizing database benchmarking results utilizing Exadata's unique capabilities. No one book can cover such a wealth of detailed information. You're simply going to need some help.

Summary

This chapter was somewhat short and covered the key hardware issues worth noting and incorporating into any database benchmarking project. Remember that we cannot simple throw hardware at a performance problem to solve it. But the corollary is also true: we cannot ignore hardware options while trying to solve database performance problems. If nothing else, just simply adding an inexpensive SSD to hold temporary or logging data can yield worthwhile results. Plus some of the newer technologies offer over 50X faster IO, with even better things coming soon. So never forget to cover the hardware aspects and options when doing database benchmarking.

CHAPTER 7

Benchmarking Software Options

The last chapter covered some hardware options important to review and consider changing when performing database benchmarking projects. While hardware advances the past decade have been nothing short of amazing (e.g., I just built a new desktop PC where the CPU offers 16 logical cores or threads), we cannot and should not always automatically look to hardware for solutions to performance issues. In reality, most benchmarking efforts can see the largest improvements to performance scores by leveraging appropriate new database features. The database vendors have likewise made significant improvements, often in response to leverage many new hardware technologies. This chapter will focus on both normal database features to consider as well as the newer ones that leverage these new hardware technologies.

You do need to keep two thoughts at the forefront of your thought process related to such new database features. First, do the benchmark specs permit using such powerful new database features, or at least not specifically state that they cannot be used? Second, does your database vendor require additional licensing fees to use these cool new features? For example, some database vendors charge extra for things such as multitenant, partitioning, advanced compression, and in-memory while others do not. In fact, one major vendor starting with their latest version

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_7

has all editions offer all features with just the maximum CPU and memory limited to what is allowed for that edition. I for one hope others follow this fine example.

Note While this chapter may highlight examples from a given database vendor, the main principles and concepts shown throughout this chapter would remain the same. So don't focus on any particular database or vendor mentioned.

Draw a Picture

When working with any database application software, and especially when doing database benchmarking projects, the first and often critical step is to construct a database structural diagram (DSD) of all the major persistent objects. Note that I purposefully did not say an entity relationship diagram (ERD) or data model since many DBA's neither like nor use them. So I hope that referencing it like this will get past any objections. When challenged on this diagramming issue, I ask those DBA's who object how they would feel if they were assembling a new bicycle for their child and the instructions had zero pictures? I strongly believe that in order to troubleshoot database performance issues, you must have a diagram since you may not have been part of that initial database design or it's simply too big to visualize in your mind.

Returning to database benchmarking, most of the specs contain such a structural diagram as shown in Figure 7-1 for the TPC-H benchmark. Note that this diagram is clearly not an ERD since it does not utilize any standard data modeling notations. At best this is a purely physical model that should be more amenable to most DBAs. But we can tell from this diagram what the tables are named, what columns they have, how they are joined, and some size estimates based on scale factor. But there

are some key items missing, such as the columns data types and which columns are indexed. So a recommendation for your consideration is to reverse engineer the database design using a data modeling tool once your benchmarking tool has created the objects. That way you'll have all that extra important information, plus you'll be able to organize the diagram layout to suit your tastes.

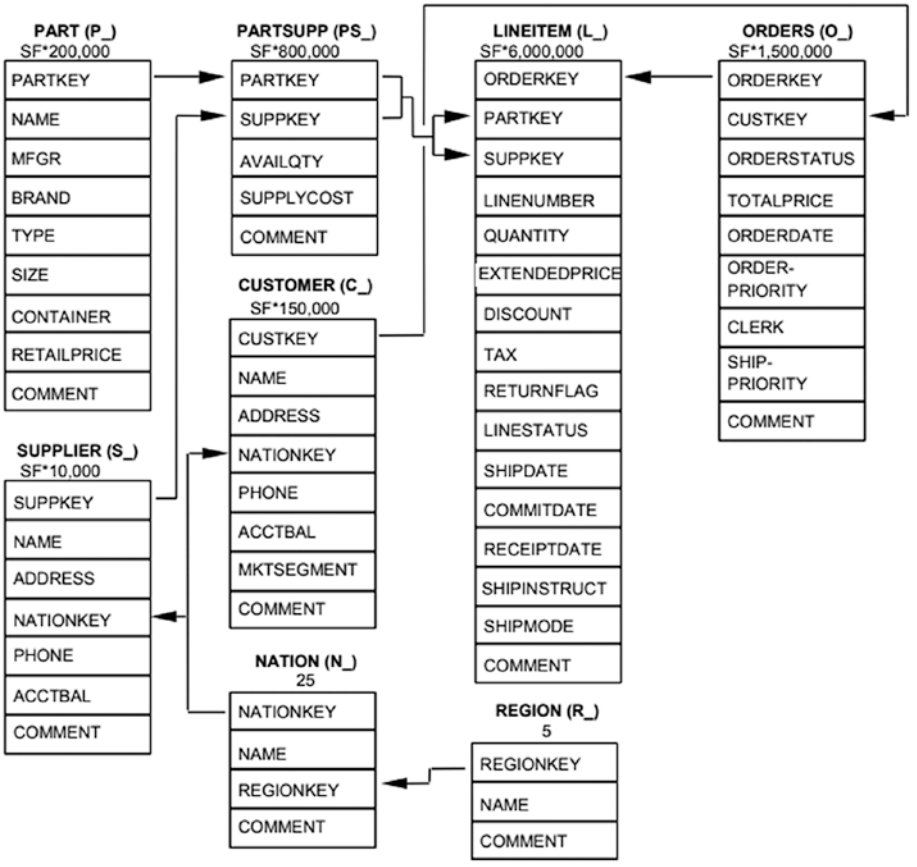


Figure 7-1. *The TPC-H Benchmark Entire Schema*

Don't let this one simple example diagram lull you into a sense of false security; the old database benchmarks were overly simple. However, the newer database benchmarks are far more complex and realistic. For example, the TPC-DS uses a star schema design that possesses seven different stars (i.e., subject areas). Figure 7-2 shows just the TPC-DS star schema for store sales. There are six other equally complex stars in the overall database design. Note that Figure 7-2 and the other six subject areas only list the table names and show the base relationship without the context of what columns are related. So once again I highly advise you reverse engineer the database design using a data modeling tool once your benchmarking tool has created the objects.

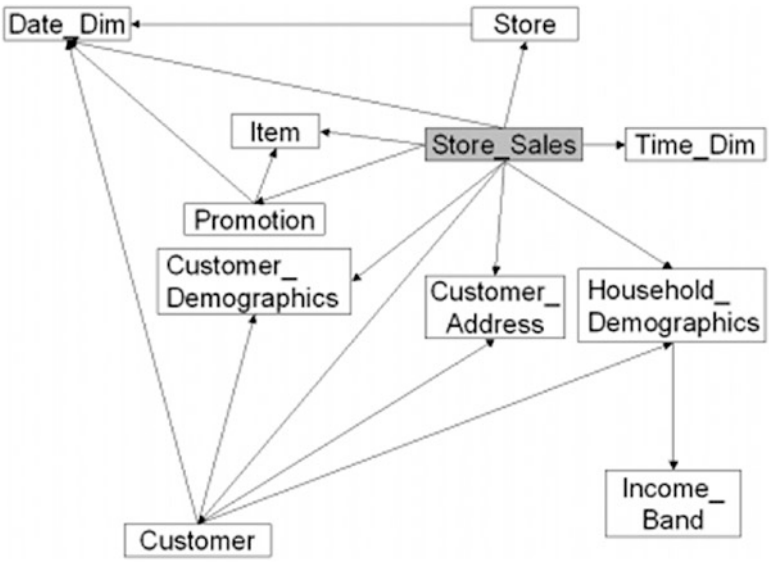


Figure 7-2. *TPC-DS Benchmark Sales Star Schema*

But the most complicated database design in my opinion is that of the TPC-DS benchmark shown in Figure 7-3. You really cannot read the table or columns names. But you can see there are 34 tables with very complex interrelationships. In this respect the TPC-DS database design

is very much like a real-world database design that you might develop for some in-house database application. Imagine looking at a TPC-DS query that's 30 lines long and which references 7 tables. Without the diagram, how can you comprehend what it's trying to accomplish let alone how those tables are really related and thus how they should be joined. Moreover, how could you suggest new indexes or optimizer hints without such knowledge? To reverse engineer Figure 7-3, I used Quest Software's Toad Data Modeler. But any good data modeling tool that offers reverse engineering should suffice. In fact, most offer a freeware that's good for some limited number of objects (e.g., 25 or 50), so you don't even have to buy a tool to build these diagrams.

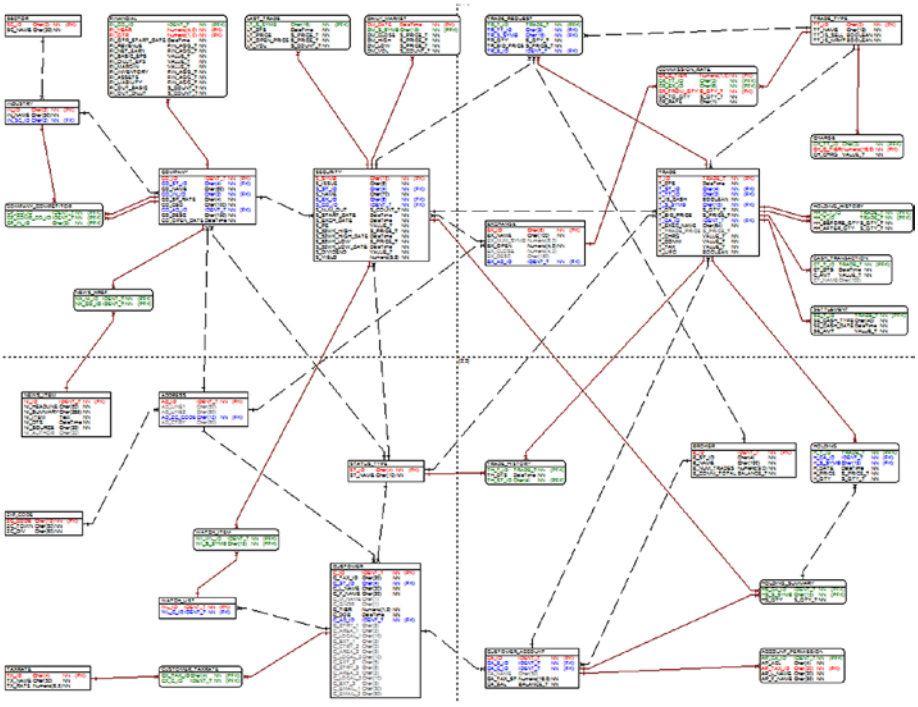


Figure 7-3. *The Complex TPC-DS Benchmark Schema*

Note The features referenced from this point on may or may not be available in your database vendor's offering, and it may be an extra cost item. So make sure to verify that you're properly licensed before using these ideas.

Partitioning Larger Objects

For very large tables and indexes, generally required by data warehousing and extremely large-scale OLTP benchmarks, the database option to partition or split objects into separate buckets that look and behave like one logical big bucket is an extremely useful technique. The idea is that queries can find rows quicker by a process known as partition elimination. Imagine a very large table called ORDERS where the data is partitioned into buckets by year as shown in Figure 7-4. Suppose we query the ORDERS table and request average order size for any order over $500 in the current and prior years. Thus the database optimizer knows it can scan the indexes just for the 2017 and 2018 buckets, and it can skip or eliminate the 2015 and 2016 index scans.

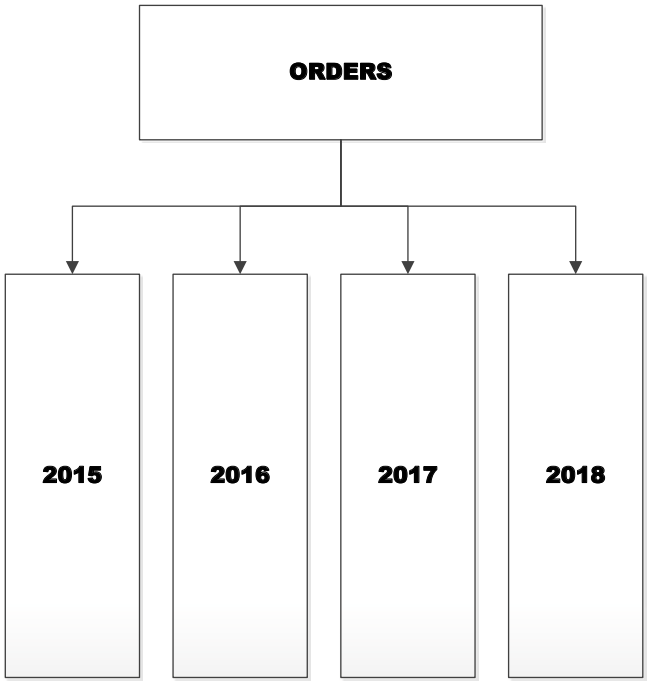


Figure 7-4. *ORDERS Table Paritioned by Year*

For appropriately very large objects, partition elimination can greatly increase query performance. However when used for smaller-sized tables the net benefit could be negligible or even less efficient for queries. So you should be careful to only use this feature when appropriate. Historically I've partitioned tables where each bucket held tens to hundreds of millions of rows. Do not take those numbers as hard advice, but rather an example to give you some idea of what very large really means by today's standards. Note too that you can also subpartition as well. So imagine that we're Amazon and have many billions of orders per year. So the bucket for a year would still be quite large. In that case you can partition the partitions: this is called subpartitioning. So in this case we might further partition each year by month. Now the optimizer can do further elimination at the subpartition level. But note in our example for any order over $500 in the

current and prior years that the optimizer cannot eliminate any of the subpartitions, so it must in fact process 12 smaller index scans rather than one large one. This could be slower, so again choose wisely.

To demonstrate an example what performance partitioning can provide, look now at Figure 7-5. The graph shows the time to run the 22 TPC-H queries against a 300GB database for a non-partitioned implementation vs. a partitioned one (where the ORDERS and LINEITEM tables are partitioned). The actual partitioning scheme is not elaborated as there's really no one universal answer as to the best ways to do it. You should partition based upon many factors, including your storage subsystem's bandwidth and capabilities. Also note that this simple method of just scoring the total runtime for the 22 queries is not the proper way to score a TPC-H benchmark, but it's reasonable and sufficient for this purpose. For my database server, database version, and configuration, the performance improvement was **35.93%**. Your mileage will vary.

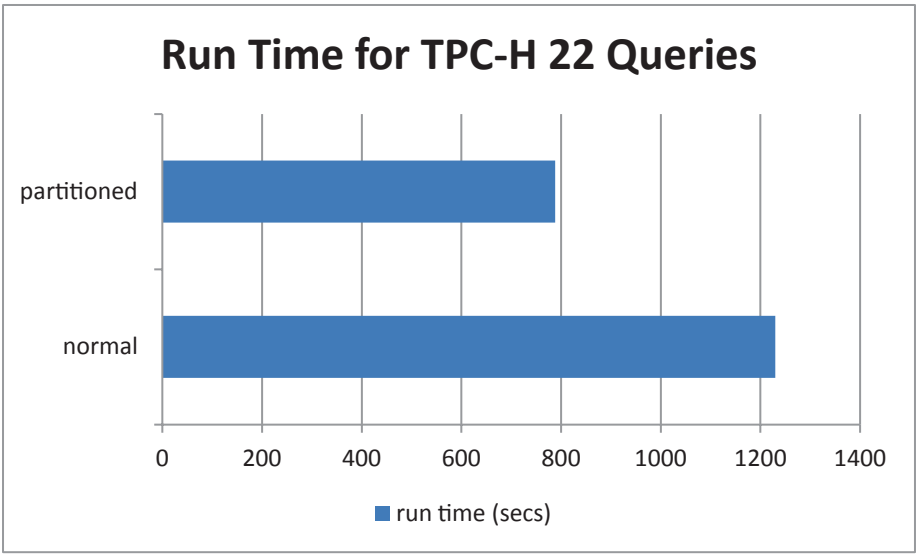


Figure 7-5. *TPC-H Improvments from Partitioning*

Advanced Data Compression

Even though the cost per GB of disk space has dropped dramatically over the past few years, there are still some reasons that data compression can benefit database performance. With the more advanced data compression that can radically shrink the data, the cost to compress and uncompress the data can be offset by the gains from reading and writing less data. Moreover the data takes less room in the database's data cache, thus the cache essentially behaves as if it were larger (i.e., had more memory). Plus if your storage is a SAN or NAS that transfers data across Infiniband, Ethernet, or Fibre Channel, the amount of data to transfer is also reduced. All of these advantages can sometimes add up to a noticeable performance improvement. But as with partitioning, you need to choose wisely when to use compression (especially if it also costs extra).

To demonstrate an example of what improvements compression can provide, look now at Figure 7-6. The graph shows the time to run the 22 TPC-H queries against a 300GB database for a non-compressed implementation vs. a compressed one (where the ORDERS and LINEITEM tables are compressed). Note that different databases offer different compression options that vary quite a bit. But compression options generally fall into three major categories:

- Uncompressed vs. compressed (simple binary option)
- Page-level compression vs. row-level compression
- Multi-leveled compression hierarchy:
    - Basic table compression
    - Advanced row compression
    - Hybrid columnar compression - low
    - Hybrid columnar compression - high

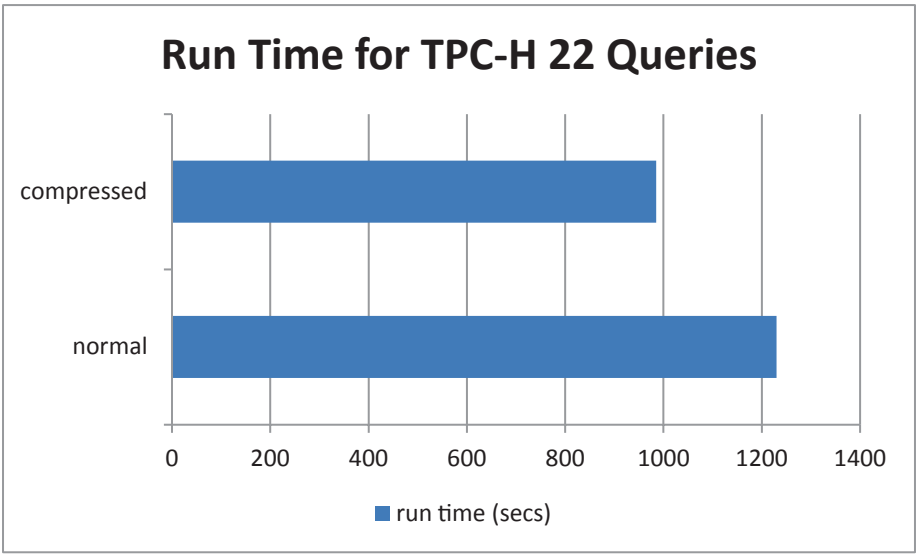


Figure 7-6. *TPC-H Improvements from Compression*

Remember that this simple method of just scoring the total runtime for the 22 queries is not the proper way to score a TPC-H benchmark, but it's reasonable and sufficient for this purpose. For my database server, database version, and configuration, the performance improvement was **19.92%**. As before with partitioning, your compression mileage will vary.

Now not every advanced database feature presented in this chapter can be easily combined for a cumulative, positive benefit; however partitioning and compression generally can be depending on your database platform. Moreover we have to be realistic regarding our performance improvement expectations. Figure 7-7 shows the net effect of combining partitioning and compression; the performance improved **46.04%**. Note that the performance improvement was not as cumulative as we might have expected. While we could have expected 36% + 20% for a total of 56%, instead we got only 46%. The reason is simple. The benefits of implementing multiple features at once generally do not result in purely additive improvements as the complex interactions between differing

techniques often skew each other. Plus there's only so much improvement that can be achieved. As the saying goes, "You can't squeeze blood from a turnip."

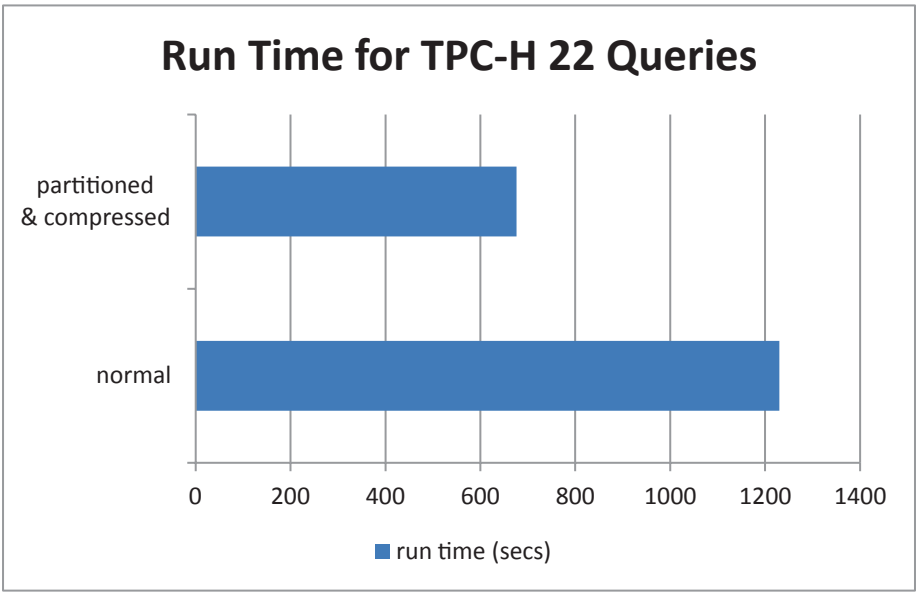


Figure 7-7. *TPC-H Improvments from Partitioing & Compression*

SSD to Extend Memory

Several database vendors offer an interesting and unique feature whereby the DBA can mount a flash disk or SSD as an extension of the database's RAM data cache. Of course, the flash disk or SSD will operate slower than standard RAM memory, but basically the entire flash disk or SSD becomes second-tier memory for the database's RAM data cache. There are two reasons this feature is very appealing. First, it's free (no added cost). Second, it's quick and easy to implement. Oracle calls this feature the "*Database Smart Flash Cache*" (not to be confused with Oracle Exadata's Smart Flash Cache). Microsoft SQL Server has a similar feature known as "*Buffer Pool Extensions.*" They both essentially do the exact same thing.

To demonstrate an example of what performance extending the memory via a flash disk or SSD can provide, look now at Figure 7-8. The graph shows the time to run the 22 TPC-H queries against a 300GB database for a RAM-only database data cache setup vs. extending that RAM via a 64GB SSD disk. For my database server, database version, and configuration, the performance improvement was only **10.33%**. You may wonder why this feature did not score better. There are several plausible reasons. First, the TPC-H queries tend to apply aggregate functions to very large collections of row data. So the cache tends to be overwhelmed. Second, there may be a point at which adding slower flash disk or SSD to the cache backfires. It's possible that scanning of mixed speed technologies for matching cache blocks or pages becomes more of a detriment than improvement. I actually got better results with a smaller flash disk (16GB). But I cannot state for sure that always going with a smaller size will be best; it just happened that way during my testing circumstances. I also got better results using a PCIe NVME flash disk, but once again not enough to say for sure to universally condemn using SSD disks for cache extension. But common sense would reasonably suggest that at the same size, PCIe NVMe flash disks should always be the best SSD disks for the key reasons highlighted in the last chapter.

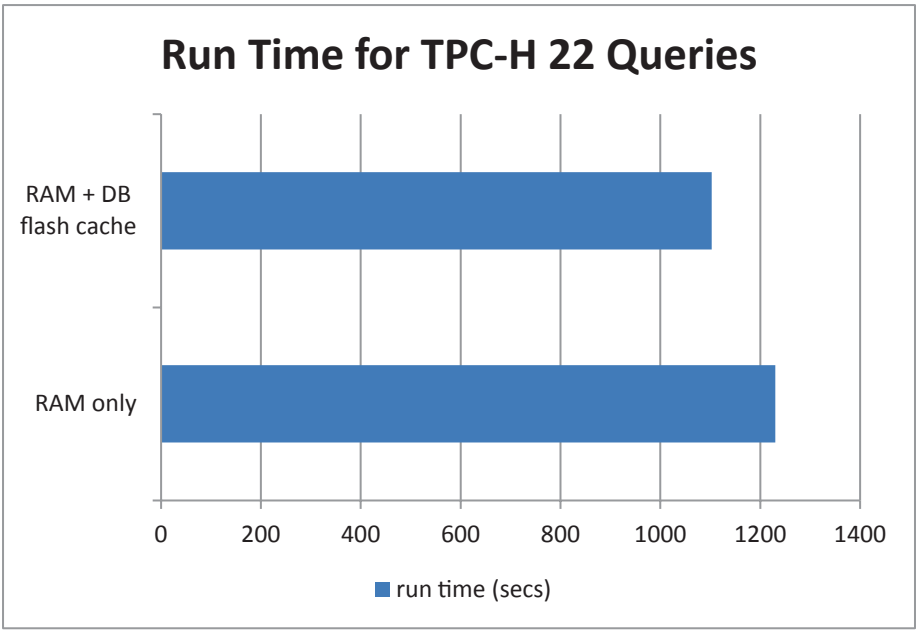


Figure 7-8. *TPC-H Improvments from SSD Extension*

Pinning Tables in Memory

Some databases provide mechanisms to suggest or even enforce that certain tables and indexes should either be pinned in memory or be aged out ever so slowly as to effectively seem pinned. You may use syntax or commands such as:

- CREATE or ALTER TABLE ... CACHE
- ALTER object_name PIN | KEEP
- EXECUTE function PINTABLE | KEEP

Another method is to allow the database data cache to be segregated into areas with different aging characteristics such as KEEP and RECYCLE. You then simply create or alter the objects to specify which of the different cache areas and thus aging characteristics to apply.

The key thing to remember is that none of these are true methods for locking a table or index in memory (that's another feature discussed later in this chapter).

To demonstrate an example of what performance pinning objects in memory can provide, look now at Figure 7-9. The graph shows the time to run the 22 TPC-H queries against a 300GB database for non-pinned setup vs. one where the smaller, lookup tables and all indexes were pinned. For my database server, database version, and configuration, the performance improvement was only **16.02%**. Once again you may have been expecting a greater improvement since pinning in memory should be really fast. But there are several reasons why this result makes sense. First, the database is really only attempting to pin the objects in memory or aging them out more slowly. So this is not a true in-memory solution. Second, the TPC-H queries tend to apply aggregate functions to very large collections of row data. So the cache tends to be overwhelmed no matter how hard you try to control it. Third, the selection of the proper objects and the aging characteristic to apply is difficult even for a schema with just eight tables (and their indexes).

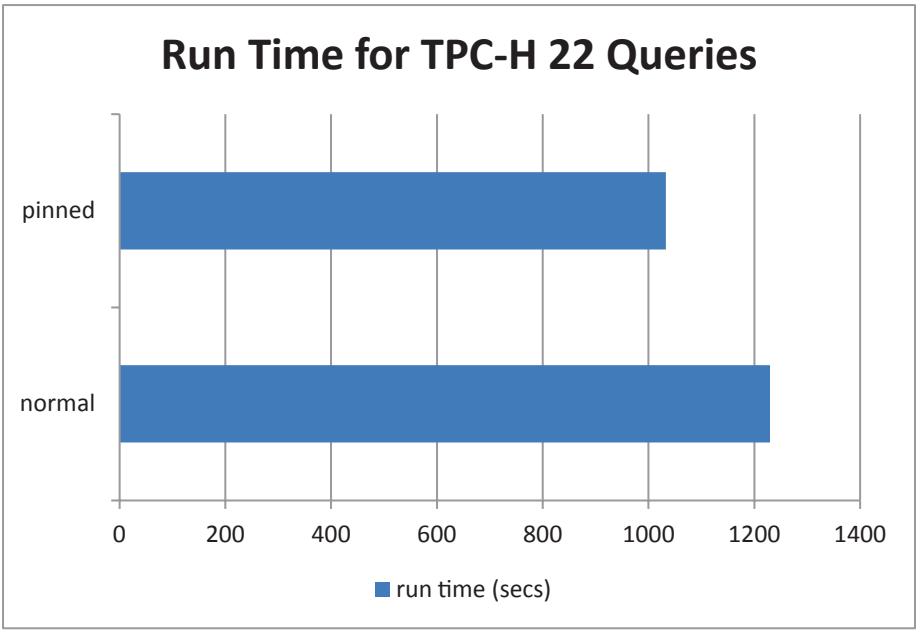


Figure 7-9. *TPC-H Improvements from Pinning Objects*

Columstore vs. Rowstore

Traditionally relational databases such as Oracle and SQL Server were based upon relational algebra and relational calculus, plus they adhered to Codd's 12 rules for any relational database. Key to this was that data was both represented and stored as tuples or rows. For many traditional business applications such as OLTP, this paradigm worked well. However, as newer business database app needs such as data warehouses debuted, the relational model did not scale as well as expected. As a result, column-oriented databases debuted, with names such as Vertica, Sybase IQ,

Teradata, Greenplum, and many others. These new column-oriented databases handled very large databases providing three major advantages:

1. Required less space (and compressed data far better)
2. Scaled from very large to extreme sizes much better
3. Reporting queries run much faster with far less IO

As relational databases began to add non-relational type features over the years such as multi-valued columns and nested tables (both major no-no's according to Codd's 12 rules), the relational database vendors naturally wanted to incorporate and benefit from the column-oriented approach. Some of these vendors simply added columnstore options while others combined in-memory (next section) with columnstore. Figure 7-10 highlights the major differences between a table stored as rows vs. columns. Note that even though these are two radically different approaches, the SQL language to work with them is unchanged. The database optimizer and engine simply can internally perform some query operations much faster when working with column-oriented data. The common example is when performing a group function on a column that's in a very large table.

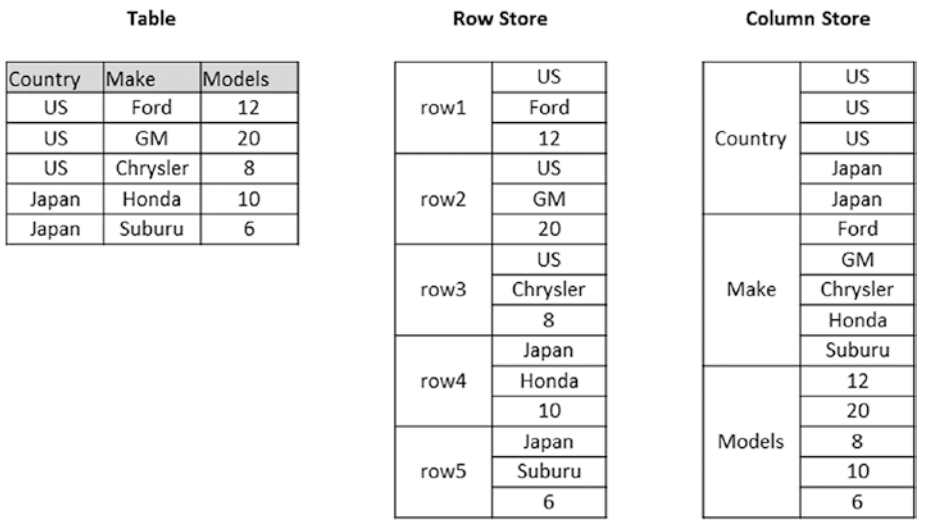


Figure 7-10. *Row Store Disk Layout vs. Column Store*

To demonstrate an example what performance a column-oriented approach can provide, look now at Figure 7-11. The graph shows the time to run the 22 TPC-H queries against a 300GB database saved as rowstore vs. columnstore. For my database server, database version, and configuration, the performance improvement was an amazing **67.48%**! That's pretty darned good considering that some of the TPC-H queries don't perform operations that benefit most from columnstore format. The database also required about 60% less disk space (without specifying additional advanced compression options).

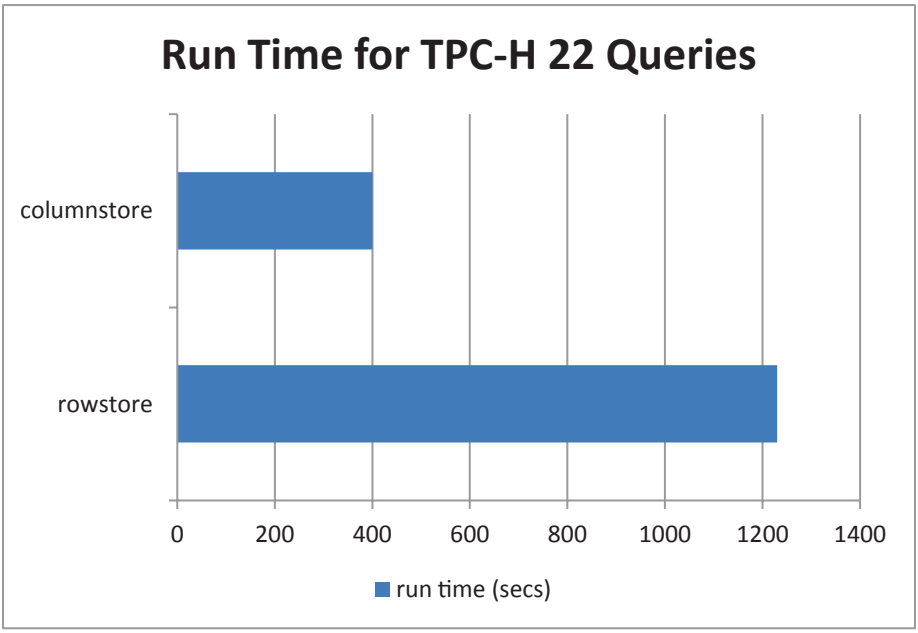


Figure 7-11. *TPC-H Improvements from Rowstore Format*

New "In-Memory" Tables

One of the hottest new technologies that several major relational database vendors have adopted is the in-memory table. Unlike the earlier section on pinning tables, which was more of a request, these new in-memory tables are truly 100% in-memory. Of course you'll need database servers with enough RAM memory for these tables in addition to all the other memory normally allocated to the database. Generally speaking, you won't be able to place the entire database in-memory, so you'll want to selectively choose those that are expected to yield the greatest return. Moreover some database vendors (Oracle) combine in-memory and columnstore as a single feature. For those databases only the in-memory tables are columnstore and the rest remain rowstore on disk. Figure 7-12 shows the complexities of Oracle's memory management.

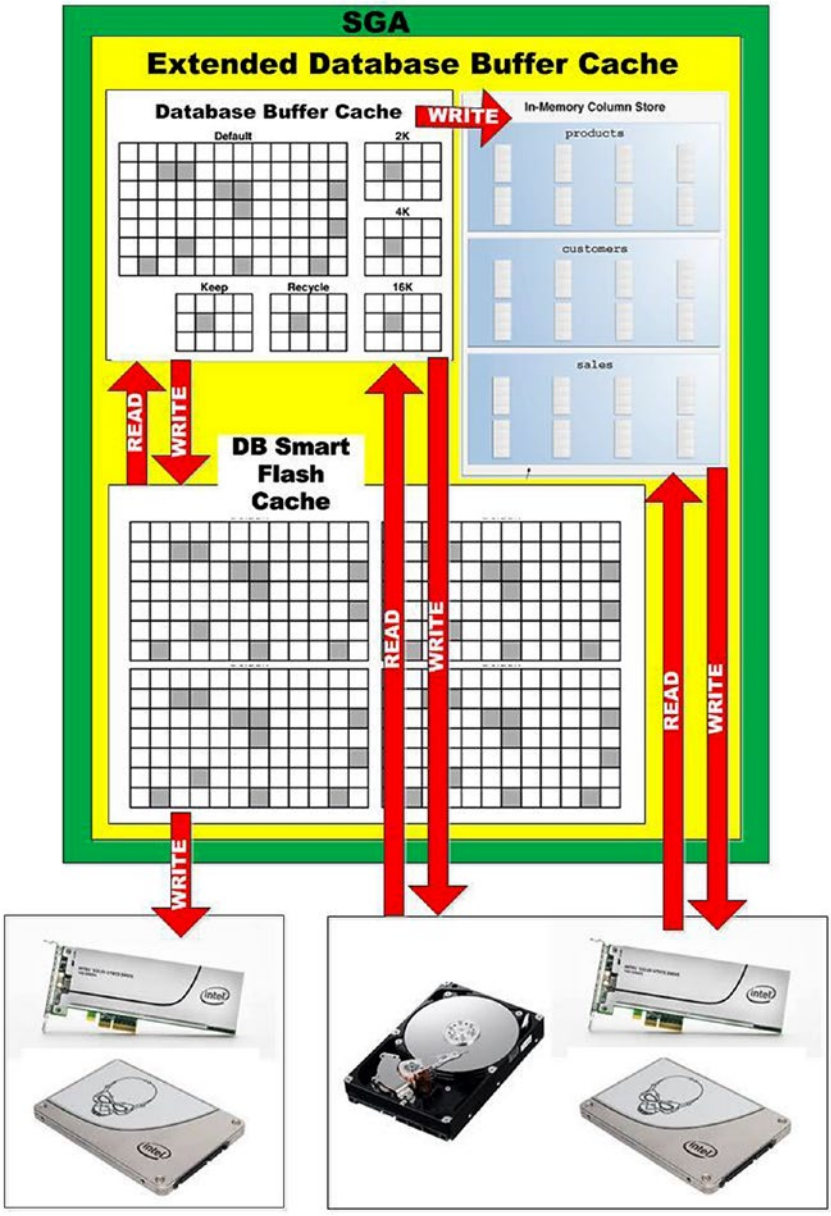


Figure 7-12. *Oracle's In-Memory Column Store Architecture*

As you can see in Figure 7-12, many of the other features from this chapter come into play. For the DBA their job of knowing how Oracle works just got a lot harder. I guess that's why they supposedly make the big money.

To demonstrate an example what performance in-memory tables can provide, look now at Figure 7-13. The graph shows the time to run the 22 TPC-H queries against a 300GB database saved as rowstore vs. columnstore. For my database server, database version, and configuration, the overall performance improvement was an amazing **90.89%**! And then when I also added advanced compression to the in-memory table, the runtime was the same and it required almost 75% less memory space! So even though my database server has only 256GB of memory, the table only needed 75GB of memory. So my database still had plenty of memory for normal database allocation and consumption. In fact, if the server only had to handle this workload, then one with less memory could be utilized. Did you catch that twist? Placing tables in-memory might possibly in some cases reduce the need for memory!

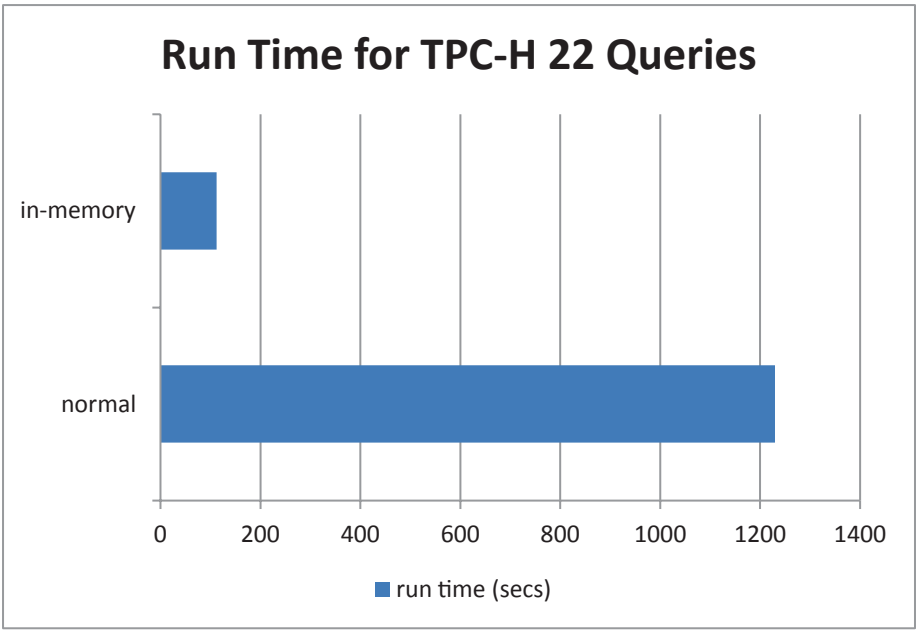


Figure 7-13. *TPC-H Improvements from In-Memory*

Summary

This chapter reviewed many database features and technologies that might be utilized when performing database benchmarks. While hardware advances covered in the last chapter have occurred at an astonishing pace, the benefits to database performance were measured. However, with recent database advances, the performance benefits have been truly amazing. Imagine queries running in 91% less time with 75% space reduction. It almost sounds like science fiction. However, the features and their results are real. Any DBA working on database benchmarking projects should strive to stay abreast of all the major database enhancements of the past few years and those yet to come. These advances currently yield the greatest benefits and sometimes with very little effort. Just make sure you're properly licensed to use these nice new toys.

CHAPTER 8

Benchmarking for Consolidation

In this chapter we're going to review benchmarking considerations for when you are testing scenarios in order to consolidate database instances. Database consolidation occurs when you attempt to reduce the total number of database licenses in order to reduce costs. With the rising cost of database licenses and the business considering a database as just a commodity, building-block object required for an application, your management might be required to undertake consolidation efforts. Moreover with today's servers being so powerful, with multi-core CPUs and lots of cheap memory, we no longer must intentionally segregate databases using the traditional "*silo*" approach of one database per server. DBAs can now realistically look to co-locate all the data from multiple databases instances under a single database instance. Sometimes such efforts may coincide with a major database version upgrade or a licensing "*true up*" effort. One thing is for sure, with all the "*database sprawl*" that has occurred over the past 20 years, we all have more database instances than anyone could have reasonably imagined. So database consolidation efforts are more common than you might think.

I also have seen consolidation efforts where management is attempting to reduce the number of database vendors they have to deal with. For example, development teams over the years may have used both Oracle

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_8

and SQL Server, so now it might be considered worthwhile to standardize on just one database vendor. I've also seen where consolidation efforts look to move from a single database vendor such as Oracle to now include a second open source vendor like PostgreSQL. The idea being to kill two birds with one stone: that is, reduce the number of high-cost licenses and consolidate on an open source platform. No matter the true goal, most database consolidation efforts can be viewed as simply combining many databases from different instances to fewer instances on more capable servers. In short, a physical database deployment reduction. So for the rest of this chapter we'll just look at this type of effort from such a one-dimensional viewpoint in order to keep things simple.

Consolidation Approaches

The first thing one must decide when performing database consolidations is by what method or approach will that consolidation be achieved. So let's begin by first picturing what databases we need to consolidate. Let's assume that we have six databases, spread across three instances, on three different severs as shown in Figure 8-1. This is a very simple yet common scenario. You might have a lot more than just three servers, but essentially it's the exact same problem just on a larger scale. Note that in Figure 8-1 we're starting on three separate physical database servers since these are legacy databases from deployment done years ago before virtualization and the cloud. So what options do we have?

Figure 8-1. *Database Instances to Consolidate*

The first option is to buy a single large physical server, have a single database instance, and place all six databases on that one server as shown in Figure 8-2. I show this option mostly because it's simple and how one would have done it in the past. But remember that today most database instances are running on hypervisors as virtual machines rather than dedicated servers. So I am not advising on the solution in Figure 8-2, but rather showing it for legacy reasons and because it conceptually or logically shows the desired end results (i.e., all six databases under a single instance and thus database license in order to reduce licensing costs). This approach sort of fits the well-known motto of the Three Musketeers: "*One for all, and all for one.*"

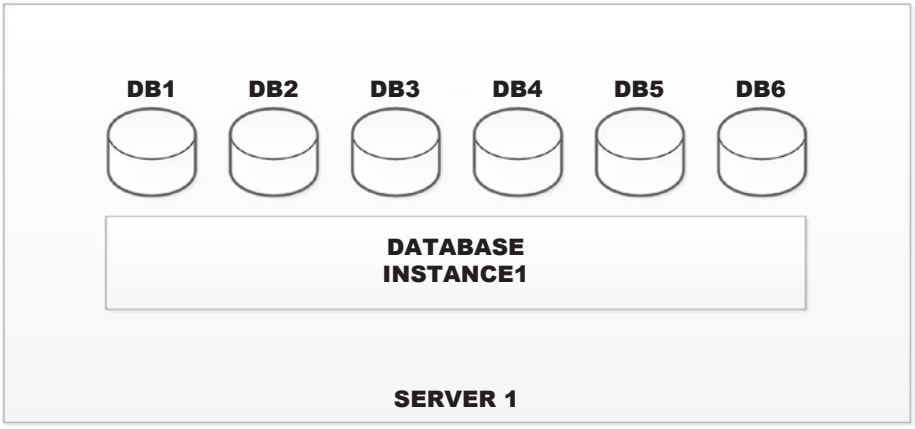


Figure 8-2. *Consolidation onto a Physical Server*

So today with the overwhelming popularity of virtualization, we would more likely transform Figure 8-2 into essentially the same basic design instead, running as a virtual machine as shown in Figure 8-3. Now you may be thinking that these last two examples were blatantly obvious and thus superfluous. But Figure 8-3 provides the foundation for what we'll be discussing in this chapter, as well as the next two chapters covering virtualized databases and databases in the cloud. Therefore Figure 8-3 should be both noted and remembered.

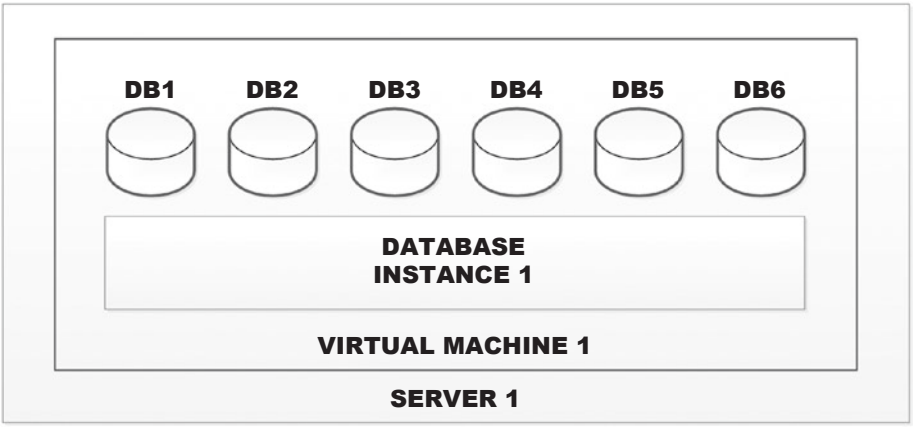


Figure 8-3. *Consolidation onto a Single Virtual Machine*

The reality is that since databases are almost universally required for any real-world application that has resulted in the very database sprawl that consolidation attempts to rectify, there are going to be far too many databases to result in either a single server as in Figure 8-2 or a single virtual machine as in Figure 8-3. You are going to need to identify the proper split where the most databases can coexist, thus resulting in the fewest number of physical servers or virtual machines. Assuming virtualization and thus building upon Figure 8-3, our first goal must be to identify the split for the right number of instances and virtual machines assuming a single infinitely scalable server can handle the

entire load resulting in Figure 8-4. We are doing this first step as a "*divide and conquer strategy*" such that once we know the proper total number of virtual machines, we can next identify the proper number of physical servers to host those virtual machines. Another way to state this is that from a scientific point of view, we're purposefully limiting the number of variables being considered such that we guarantee that we don't introduce uncertainty to the conclusions.

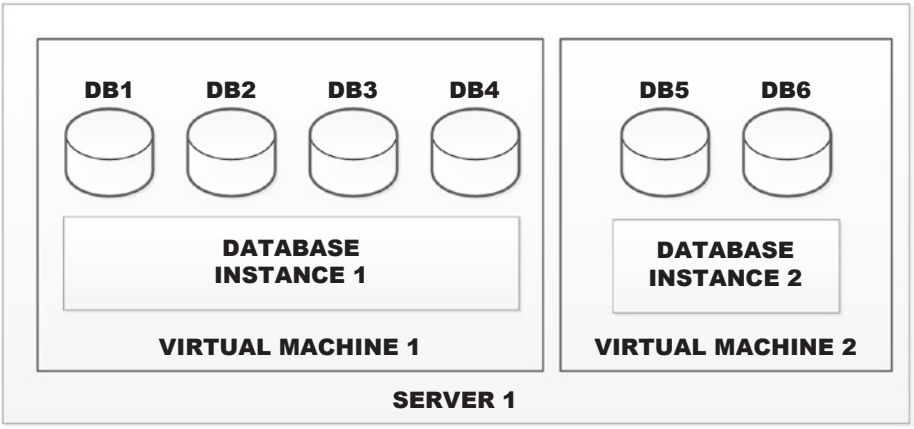


Figure 8-4. *Consolidation onto Multiple Virtual Machines*

Once we know which databases can peacefully coexist on a single virtual machine because their workloads are reasonably compatible, meaning that the workloads don't make net requests that cumulatively overly stress the available resources, we can then see how many actual servers of what size are required for each virtual machine and its collections of databases. Moreover, we can ascertain whether multiple virtual machines can be hosted by a server. For example, in Figure 8-4, if we know that virtual machine #1 and its databases require 2X resources and virtual machine #2 requires 3X resources, then we require a server with 5X resources or separate physical servers to host them. That's what we'll be attempting to benchmark in this chapter.

Database Coexistence

The first step is to find which currently separate database instances and their databases can coexist on a single virtual machine. Let's assume that the current separation across physical servers is correct or at least is justifiable based upon compatible workloads. Then what we are trying to discover is using Figure 8-1, can we combine physical servers #1 and #2 along with their two instances and four databases? To accomplish this, we need to benchmark as shown in Figure 8-5. This diagram is rather complex, so let's examine the three major pieces. The bottom portion of this figure coincides with the left half of Figure 8-4, finding the databases that can coexist on one virtual machine. The top portion of this figure represents the desktop or laptop PC where you have installed database benchmarking software for running the central command console. The middle portion of the figure represents the agent or agents required for each currently running instance of the benchmarking central command console. None of the currently available benchmarking tools offers the ability to easily run multiple tests at the same time, at least not easily nor reliably. So if we need to run two concurrent database benchmarking workloads in order to correctly simulate the applications previously hitting separate physical servers, then we'll need to run two separate benchmarking consoles. Furthermore, based upon the benchmark scale factor and number of concurrent users' sessions, each central console may require multiple benchmarking agents to run the cumulative workload. In fact, you may even need to spread the agents across virtual machines and possibly across physical servers in order to handle the resources required to run the entire benchmark workload.

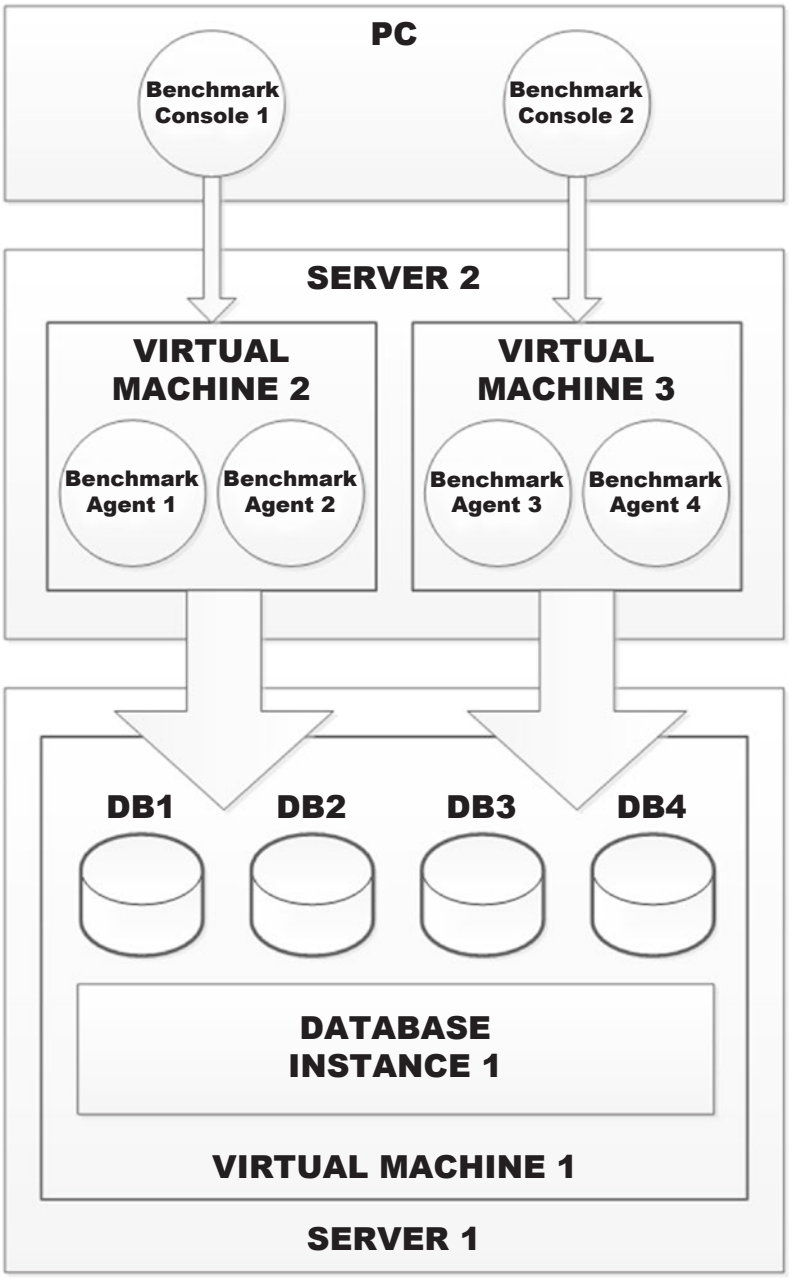


Figure 8-5. *Benchmarking Setup for Consolidation*

There are a couple of management issues required by the architecture shown in Figure 8-5 that are probably not readily obvious. First, you will need to manually coordinate all the efforts across two separate benchmarking consoles. There will not be any automatic or synchronized start and stop for the running of database benchmarks. Second, all the logging for warnings or errors and for runtime performance results will be separate and non-correlated. So once again there will be manual efforts required to dissect and divide what you have learned for a given benchmark test. Third, you are probably going to need additional physical servers and virtual machines than originally considered, so there may be both measurable time and effort required to assemble the proper database benchmarking architecture as shown in Figure 8-5. For example, I worked on one project where we required 300 separate agents across over 100 virtual machines and a dozen physical servers in order to generate the proper workload to adequately stress the candidate, consolidated databases. So plan accordingly.

Unforeseen Issues

When attempting to combine database instances and their databases, some unforeseen issues can arise. Imagine the DBA starts by setting an instance configuration setting by simply adding that setting from the two instances. So, for example, if instance #1 required 100GB of memory and instance #2 required 50GB or memory, the DBA might configure the single consolidated instance at 150GB of memory. However, it's been my painful experience that nothing in life, especially not database instance configuration parameters, is quite that easy. You're going to need to do your homework, most likely reviewing all the key database instance configuration parameters. It's possible, depending upon your database platform, that the proper, net setting could be either more or less than the simple sum of the values. In some cases, such as when the value

should have been larger than the sum, it could well skew the database benchmarking results.

Another issues that often rears its ugly head is the storage situation. We have been focusing on server consolidation without regard to the IO, yet we all know that the Achilles' heel of any database is physical disk IO. The key issue, assuming that the storage arrays and LUN's remain the same is IO bandwidth. Can the virtualization server running the hypervisor offer at least the maximum net IO throughput required by all the consolidated databases, with extra being desirable for atypical peak loads? We'll cover some specific issues pertinent to achieving this goal in the next chapter on benchmarking virtualized databases. For now it's just a noteworthy item for consideration when consolidating. The basic question for now is does physical server #1 possess a sufficient number of host bus adapters (HBAs) or network interface cards (NICs) with proper multi-path configurations in order to handle at least two times the probable maximum concurrent IO requests? If not, then it's very likely that such an IO bottleneck will occur during database benchmarking efforts since the scale and concurrent user sessions will be purposefully specified to stress the maximums. The result will of course be inaccurate results. So make sure to check the IO bandwidth.

Server Distribution

Once you establish which database instances and all their databases can be consolidated, the next logical question is which, if any, aggregated instances can be combined on the same physical or virtualization server? Let's assume that you are either virtualized or in the cloud, then the importance of this issue is greatly reduced because these instances can either be relocated or their virtual machines augmented as needed. For example (returning to Figure 8-4), if we know that virtual machine #1 with its instance and four databases requires 5X resources (as calculated earlier

in this chapter), and then we next determine that instance #2 and its two databases require 4X resources, then we know that we need a physical server that can provide at least 9X resources. But once again we need to consider intangible issues in consolidating, which might tend to skew the simple calculations performed in isolation. What I am suggesting is that if the calculation yields 9X, that interpreting that to mean 10X or slightly larger when combined will be the safe bet. Again, this may appear like a lazy oversimplification, but with dynamic virtualization Vm relocation and the cloud offering resizable VM sizings that cover a wide spectrum of possibilities, the need to nail this one is far less important than the simple consolidation step.

Mixing Benchmarks

When looking to consolidate database instances and their databases, you can almost bet money that the databases will be different in nature and purpose, and thus their workloads will also be quite different. It's highly unlikely that you'll be trying to consolidate nearly identical natured databases in all cases, thus you will most likely need to tailor your database benchmark tests to best approximate the transactional nature and mix you're attempting to consolidate. Sounds simple enough. The problem is that, once again, most of the currently available database benchmarking tools generally do not offer an easy way to construct your own database benchmarks by combining the existing ones with different weightings or percentages. So you end up with an even more complex architecture as shown in Figure 8-6. Imagine you want to run a mixture of 50% of a TPC-C and 50% of a TPC-H per database. You'd have to have a central console running for each benchmark now, or four total benchmarking central consoles – once again having to manually manage all the timing, dependencies, and interactions between them. This is basically too unwieldy for the average benchmarking user. There simply has to be a better way.

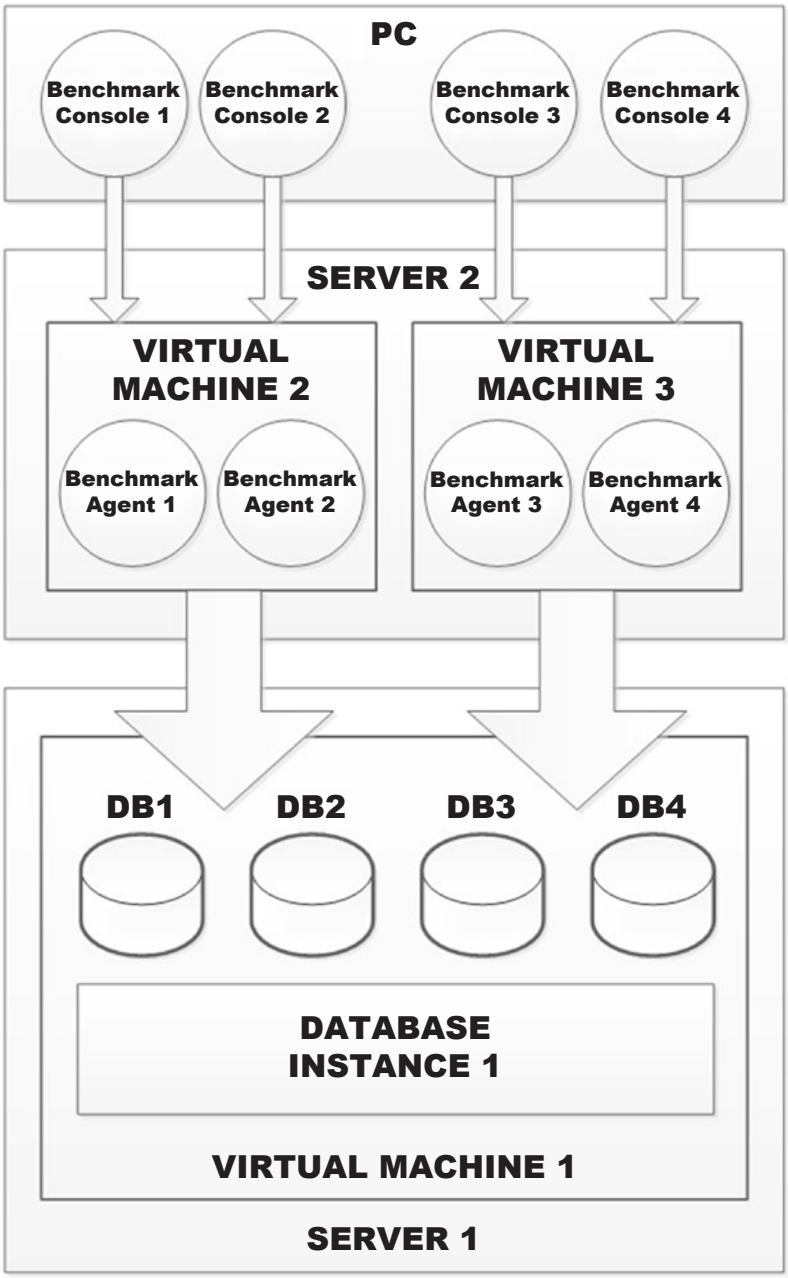


Figure 8-6. *Mixing Database Benchmark Types*

I've purposefully restrained from recommending any one database benchmarking tool for a host of reasons. First, none of these tools is perfect and can be your one and only solution. Second, I don't want to sound like a vendor schill by endorsing any solution. But that said, I can honestly say that for this kind of concurrent, mixed-database benchmarking execution, scenarios that one tool does offer a feature to make this need far more tenable. That tool is Quest Software's Benchmark Factory (BMF). It has a very simple and useful wizard which enables the user to construct their own custom database benchmarks as shown in Figure 8-7, including allowing for meaningful combinations from among the following:

- Industry-standard database benchmarks
- application transactions (OLTP vs. DW)
- SQL statements (SELECT, INSERT, UPDATE, and DELETE)
- Hardware stressing (CPU, Memory, Disk IO, and Network)

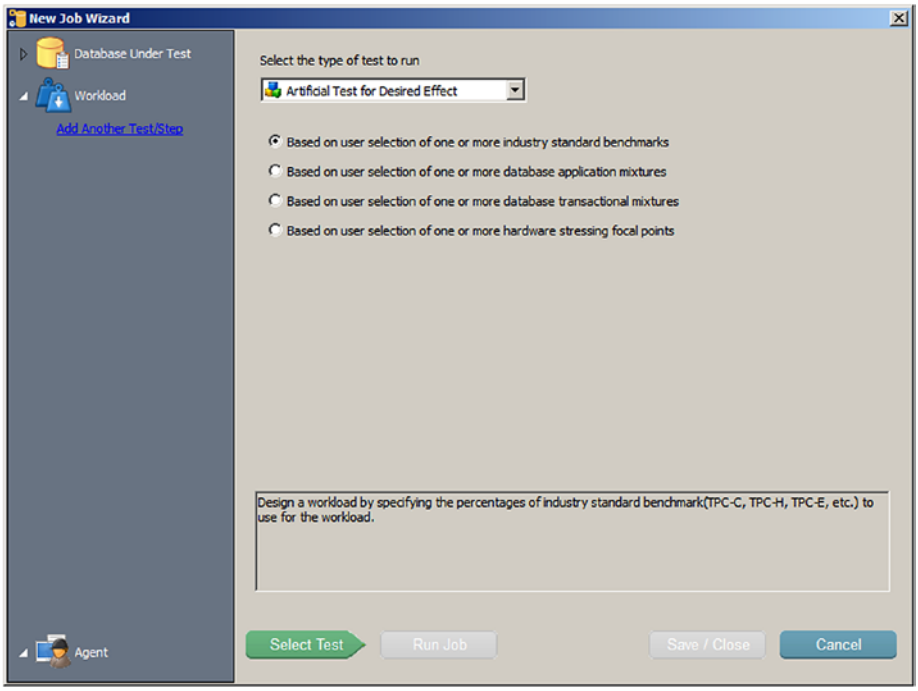


Figure 8-7. *BMF Allows User-Customized Benchmarks*

Let's examine the two most valuable of those four options: application transactions and hardware stressing. Figure 8-8 shows the very simple but highly useful application transactions option where you simply specify what percentage of OLTP type transactions vs. the percentage for a data warehouse. Next you provide a scale factor for each type of test and you're done. BMF will then create a random mixture of those types of transactions in those percentages and run them all in an intermixed order. With this one feature alone, most users can quickly and easily construct the database benchmarks they require for effective database consolidation.

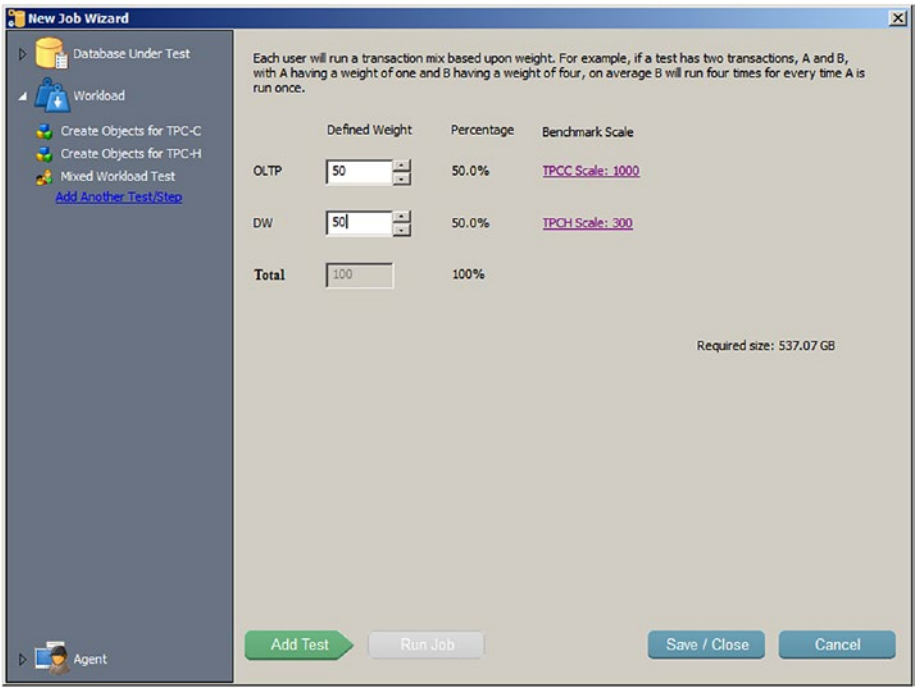


Figure 8-8. *BMF Offers Mixed Transaction Workloads*

Figure 8-9 shows the second, also simple, and useful hardware stressing option where you simply specify what percentage of stress to apply to each hardware resource. Next you provide a scale factor for each type of test and you're done. BMF will once again then create a random mixture of those types of transactions in those percentages and run them all in an intermixed order. I would argue that the former custom test is better suited for consolidating databases and the latter for calculating server distribution (both topics previously covered in this chapter). However, you may well find both useful for each stage.

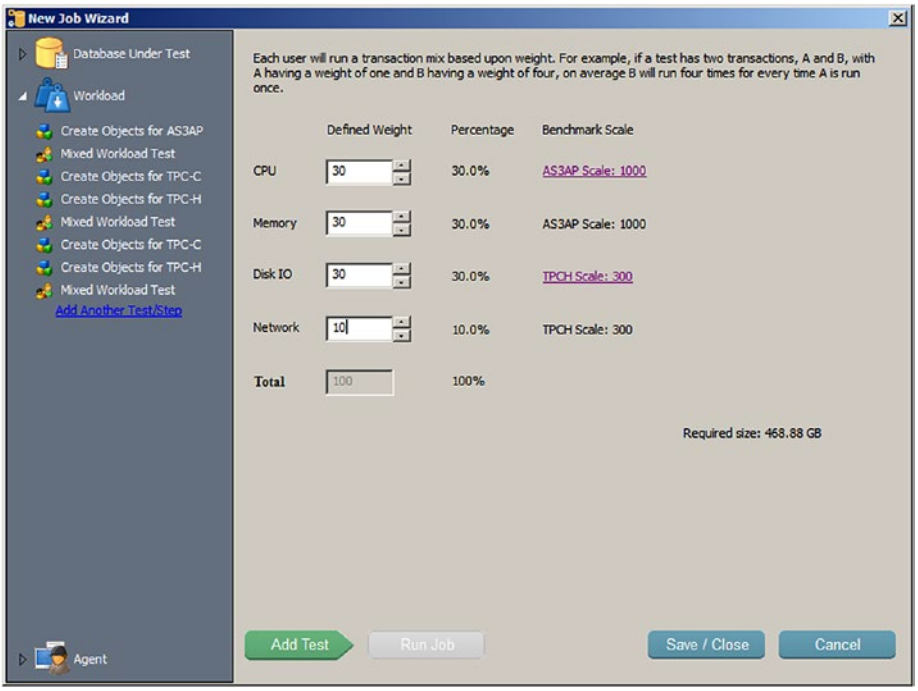


Figure 8-9. *BMF Offers Hardware Stressing Workloads*

Summary

In this chapter we examined the issues and techniques for performing database benchmarking for the purpose of supporting database consolidation efforts. We assumed that databases being consolidated would also be virtualized for several reasons, including dynamic relocation of virtual machines and virtual machine or image sizing for the cloud. This chapter also stressed the special challenges required to test multiple databases concurrently and the complex architecture required by most of today's benchmarking tools. Finally, we examined one tool, Benchmark Factory, that offers some key functionalities that make database consolidation testing efforts far easier.

CHAPTER 9

Benchmarking for Virtualization

In this chapter we're going to review benchmarking considerations for when you are benchmarking scenarios in order to virtualize database instances and their databases. In many respects the prior chapter on database consolidation is a prerequisite for this chapter. That chapter's benchmarking efforts to identify which databases can share an instance in order to reduce the instance count, and then how many servers are required to host those instances are much the same. As such, we'll therefore consider the prior chapter the first past part of the virtualization effort. This chapter will then deal with the second part: are their issues pertinent and specific virtualization that can affect database benchmarking efforts and their results.

In this chapter the examples shown will be from VMware's hypervisor. Again, I am not suggesting that you choose any particular vendor nor endorsing any vendor. In this case I am showing VMware for two reasons. First, I happen to have a VMware ESX server handy here in my home lab. Second, VMware has a majority market share and thus many, if not most, DBAs are somewhat familiar with it. I also have regularly utilized VMware Workstation, VMware Player, Oracle Virtual Box, Oracle VM (a Xen-based solution), QEMU, and Linux KVM. I mention this mostly to reassure the reader that I have sufficient exposure to multiple vendors, and thus I know

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_9

that the concepts presented in this chapter have relevant and similar counterparts across most virtualization vendor solutions.

Note While this chapter may highlight examples from a given virtualization vendor, all the principles and concepts shown in this chapter would apply for any virtualization vendor's solution. The exact name or terminology might change, but the basic concepts would still apply.

Server BIOS

The first and most obvious thing that DBAs need to check before benchmarking is whether the physical server's BIOS has been properly configured for running any hypervisor. You might assume that any server is shipped from the factory with the recommended setting shown in Table 9-1, but that's not always true. Plus what if the server has been repurposed or was purchased used; in either case, once again, these settings should be checked. I have seen one or twice where these were not set, were assumed set, and later discovered during our benchmarking efforts when the results were either below expectation or not repeatable. So spend the five minutes during a server boot to check the BIOS settings just to be sure.

Table 9-1. *BIOS Settings for Virtualization*

BIOS Setting	Recommend
Virtualization Technology	YES
VT-x, VT-d, AMD-V, AMD-Vi	YES
Node Interleaving	NO
Turbo Mode	YES
Hyperthreading	YES

Note Some DBAs may remember that in the early days of Hyperthreading that it was not effective for databases. However, Intel has long since rectified that initial issue. But time permitting, you might still want to benchmark with it off just in case your database is an exception to the norm.

VM Configuration

When creating the virtual machine image definition to host their databases, many DBAs might focus on the CPU and memory first, with some thought then also to the disks allocated. But there are other important settings to be aware of and many levels of options for many of these settings. This book is not intended to provide a complete, best practices for deploying databases on a hypervisor, but we will cover the key ones in this chapter that are proven to have a significant impact on database benchmarking.

CPU

The first and most important CPU guideline is to not overallocate processors. This is a hard habit for DBAs to break since for decades, on physical servers we generally had to overorder to account for growth and unpredictable peak workloads. So it's sort of ingrained now in our nature. But with virtualization we can dynamically add CPUs when needed, often without a reboot depending on the VM operating system and database. Plus virtual machines can also be relocated to larger servers or those with less total workload (and therefore have extra CPU cycles to leverage). Finally remember that all virtual machines must share limited resources, including CPUs, so we should not overallocate or we'll soon end up with just one or two virtual machines per physical server.

Another reason not to overallocate CPU resources is that sometimes it can actually not provide any net benefit. For example, let's assume that you're running SQL Server standard edition, which has a limit of the **lesser** of 4 sockets or 24 cores. So when you allocate the virtual machine virtual CPUs, you can easily choose a suboptimal value without realizing it. In fact, Brent Ozar, a world-renowned SQL Server MVP and expert, has written blogs and papers on this very topic. Look at Figure 9-1; this virtual machine, which is running SQL Server, has its CPU count defined as 12, meaning 12 sockets, with one thread per CPU socket. Therefore SQL Server will not use all 12, but instead cap its CPU usage at 4. What the CPU setting should have been set to is 3 with 4 threads per socket. This is a very easy mistake to make and to overlook when diagnosing poor performance. SQL Server is not the only database with such issues, I just used it as an example to make the point. This is probably one of the quickest and easiest things to correct to improve virtual machine database performance. Your virtualization administrators may not be aware of any such database limits, so this mistake happens more than you might suspect.

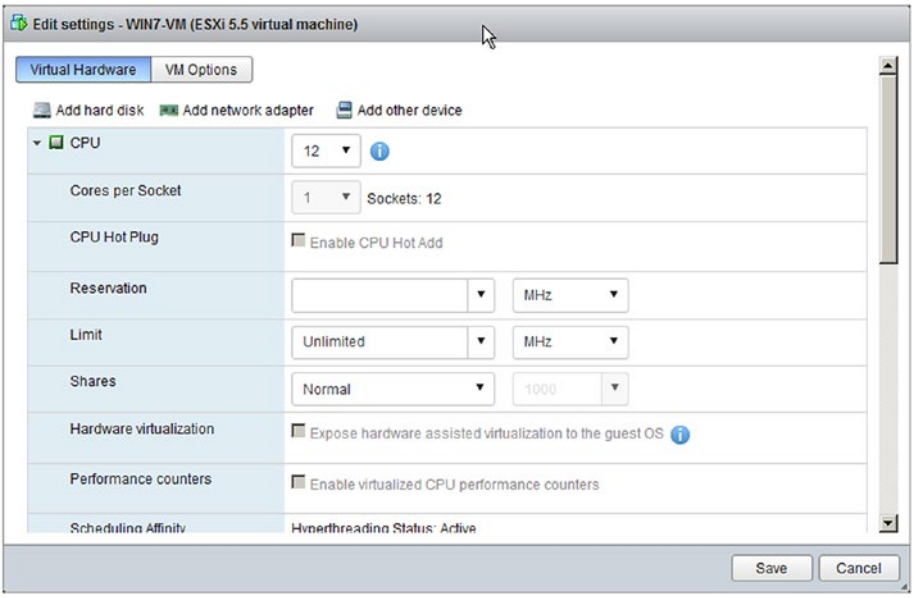


Figure 9-1. *Inefficient VM CPU Configuration*

The final database virtualization issue I'll make is not to overassume hyperthreading effectiveness. It's easy to buy the hype and think it means 2X the CPU power since the count doubles. The consensus quoted by many DBAs across the web is that hyperthreading seems to realistically and reliably yield about 10 to 30% improvement. I suspect that there's some leftover skepticism from earlier versions of hyperthreading, so my personal recommended rule of thumb is to count hyperthreading as 50–60%. If you're not overprovisioning, this value should be safe and work well enough for most cases. But you'll have to decide for yourself what value to ascribe. Just realize it would be imprudent to assign a value of 100% (or even close to it).

From a database benchmarking perspective, the VM CPU setting and its effectiveness can yield a dramatic improvement. Let's assume you're running a TPC-C with a large-scale factor (e.g., 1,000 warehouses) such that you can run 10,000 concurrent user sessions. Clearly the virtual

machine's CPU resource allocation is important when you're running thousands of sessions. But let's instead assume that you're running a large scale TPC-H or TPC-DS; however with far fewer user sessions, you might then consider the CPU setting as less important. But you'd be wrong! These large data warehouse workloads run extremely complex queries often against partitioned tables. Most database optimizers will attempt to subdivide and parallelize the execution of inter and intra partition operations, as well as for large table and index scans. So this setting can be critical here as well.

Memory

The second and equally critical area is virtual machine memory allocations. A lot of times DBAs just assume they can use golden rules from days past. That's kind of true, but with some very important twists. When deploying a database on a physical server the memory available to that database was just the amount the server contained. The DBA would then do some rough calculations along the lines of the following:

- `Total VM Mem = Total DB Mem Allocation + (# DB processes + # OS Processes) * 2MB + OS Mem + VM Overhead`

However, for a virtual machine you must also incorporate the way the hypervisor manages VM memory. Some hypervisors require a driver installed on the VM, which permits the hypervisor to steal idle memory from one VM to serve another. VMware calls this the "*balloon driver.*" However the hypervisor obviously cannot borrow all the memory, so there is a limit (or floor). You can set that limit for the virtual machine, and it is often referred to as reserved memory. You can think of it as the absolute memory floor below which the hypervisor cannot borrow further, and is set like that shown in Figure 9-2. So in this case I did the calculation like the one above and the number came out to 16GB. When I then exclude

the last two items (i.e., OS Mem and VM Overheard) the value is 12GB, so that's what I set the reserved memory to. This memory setting should significantly reduce the probability of both ballooning and VM OS swapping, thereby guaranteeing that the VM reserves the memory it needs for optimum performance. Note that the reserved memory plays one other significant role. Let's assume that your hypervisor allows for dynamic VM relocation, then the new host would have to have at least the VM's reserved memory amount free or the VM could not be migrated to it.

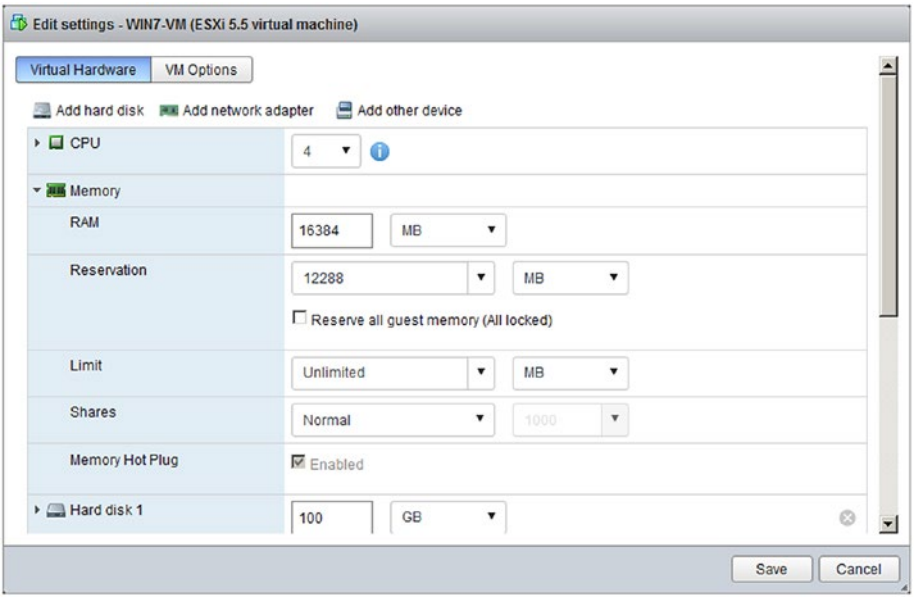


Figure 9-2. *Setting a Borrowed Memory Floor*

From a database benchmarking perspective, the VM memory setting can also yield impactful improvements. But the reason is related more to the nature of virtualization than from a database perspective. If all or most of the virtual machines overcommit memory, then the hypervisor might need to perform swapping. The can be very costly because the hypervisor does not know the internal VM OS context of its memory usage, thus the

hypervisor could select memory pages to swap that are suboptimal for the guest OS. It could swap out critical VM memory pages that the guest OS would not, such as the Linux kernel. The hypervisor could also choose to swap out memory pages that are clean at the guest OS level (i.e., empty). Finally, a condition referred to as double paging can occur, where both the hypervisor and guest OS swap out the same page, which then cause problems when brought back in.

Network

There are just two very simple issues and recommendations involving network settings for a virtual machine undergoing any database benchmarks. First and foremost, the network adapter should be set to the proper and optimal driver. I've been involved in a handful of benchmarking efforts where the expected results were not being met and no one was quite sure why. Upon investigating their VM network settings, as shown in Figure 9-3, a suboptimal driver had been designated. When the VM was either created or converted/migrated to the host server, the network driver was set to the common default for the guest OS (in this case, E1000). However, for this hypervisor (i.e., VMware ESX), the best driver was a paravirtualized driver known as VMXNET3. Once again, a very simple and easy thing to check and then correct. Remember that this database VM for a large OLTP benchmark could be handling hundreds of thousands to millions of user session requests and returned results. For a data warehousing benchmark, it could be returning fairly large amounts of data. Either way the network will be a serious constraint on overall performance. In the case of an OLTP test this simple setting change made a 500% improvement in average response time, which was the factor we were most interested in optimizing. If you're wondering how that could be, the E1000 driver is software emulation for a generic 1 gigabit NIC, whereas the VMXNET3 is for 10 gigabits. Why does this happen? The virtual machine must have the vendor's tools software installed in the guest OS for

the option to be available. So until you manually install the vendor's tools, the network driver is set to the suboptimal default.

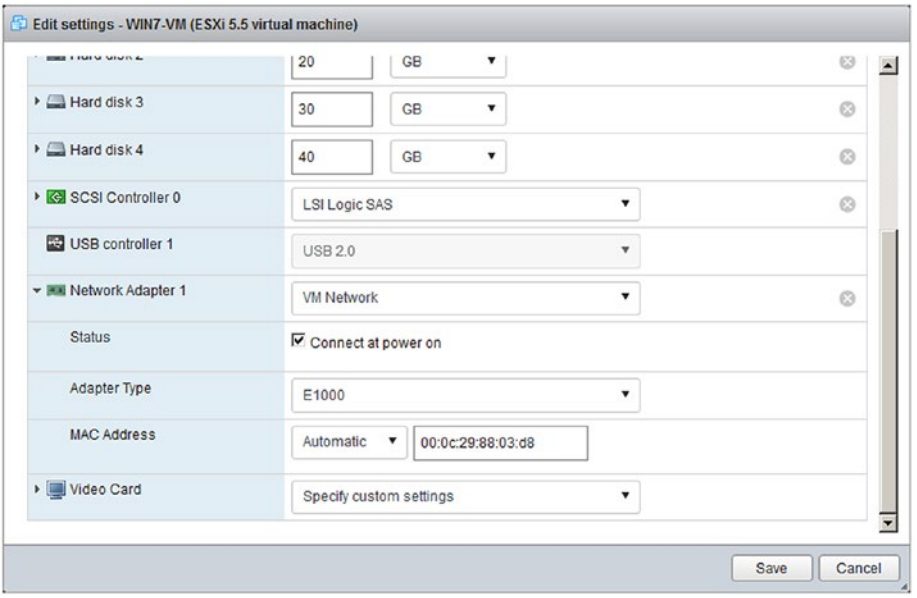


Figure 9-3. *Choose Optimal Hypervisor Network Driver*

The second networking consideration is if your hypervisor permits "*passthrough*" mode for a physical device, then you can choose one of your network interface cards and mark it as passthrough enabled. Then you can assign that NIC to the VM such that the guest OS sees the actual hardware and therefore uses the native OS hardware driver – which also means your network traffic will entirely bypass the hypervisor and its overhead. However, there is one major drawback to this using the technique: dynamic virtual machine relocation might be impossible or far more difficult since servers could have different hardware or all that new hosts hardware could already be provisioned such that nothing is left for any new passthrough setups. As such, you may well conclude that for network traffic that the optimal hypervisor network driver (e.g., VMXNET3) is

sufficient in order to retain the most overall flexibility with reasonably acceptable performance. I've personally done this both ways and cannot really make any recommendations. Remember that the hypervisor host may have a Link Aggregation Control Protocol (LACP) setup combining the throughput of multiple NICs that you'd miss out on by doing your own dedicated passthrough on one or two NICs. For example, my server has eight gigabit NIC's, of which six are allocated to the LACP group.

Disk IO

As has been said several times in this book, we all know that physical disk IO is the Achilles' heel of database performance. Remember the key points from earlier:

- Avoid RAID 5 for database data and index files.
- Suggest RAID 10 for database data and index files.
- Storage LUNs with proper stripe width and depth/size.

So what considerations do we need to add to this list when doing benchmarking for virtualized databases? There are just three additional, and they are all pretty straightforward:

- Avoid virtualization snapshots (adds unnecessary overhead).
- As with the network, consider passthrough drivers for host bus adapters (HBAs) when reasonable (just as with networking this choice may preclude dynamic virtual machine relocation).
- Choose the most efficient disk initialization mode for optimal database IO.

It's this last item that requires explanation. Many DBAs would agree that for those database platforms that allow for database data files to grow dynamically, the cost of dynamic space allocation can be a measurable burden. For example, imagine a database file set to start at 50MB and to grow by 50M as space needed is being utilized to house the schema for a 300GB TPC-H. The data loading process will cause roughly 600 dynamic disk space extensions just for the data, with maybe another 100 for the indexes. The smart DBA would just create a data file with maybe 500GB of disk space and avoid all that dynamic allocation overhead. The same principle holds true for your virtual machine disk allocations. For example, VMware offers three provisioning modes:

- Thin: file starts at zero size and dynamically grows as space requested.
- Lazy Zero Thick (aka Flat): file starts at the specified size and blocks must be dynamically zeroed out before written.
- Eager Zero Thick: file starts at the specified size and all blocks pre-zeroed out before file made available for use.

So when database benchmarking and looking for the best disk IO possible, you should not only minimize database-level dynamic disk growth but also complement that by eliminating hypervisor-level dynamic disk growth by choosing the entirely preallocated mode as shown in Figure 9-4.

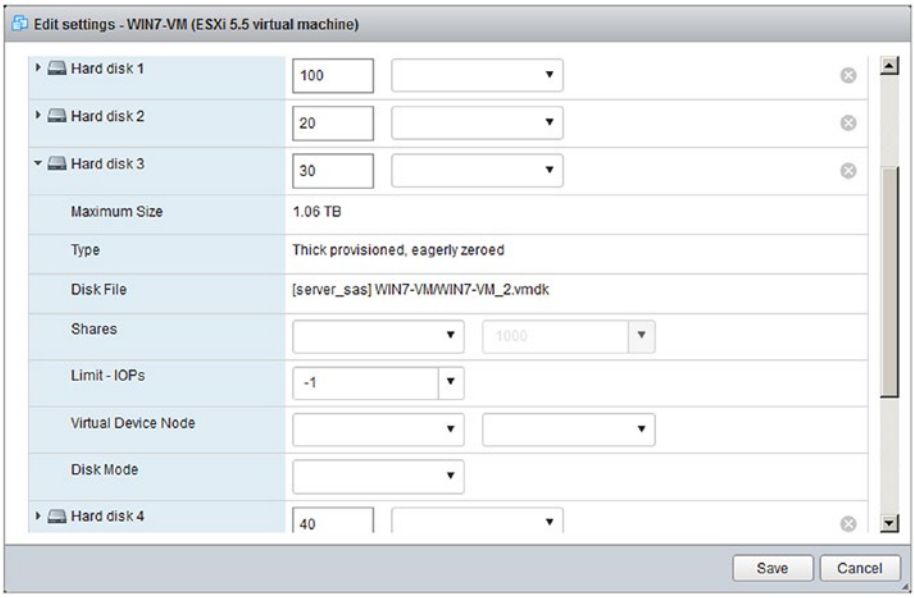


Figure 9-4. *Choose Optimal Disk Provisiong Method*

Monitoring Issues

Back in Chapter 5 we visualized the database benchmarking effort as being similar to a cardiac stress test as shown here again in Figure 9-5. We've not made much mention of the monitoring aspect throughout the book, until now that is. It's a fairly well-known fact that not all OS and database monitoring tools report accurate values when the guest OS and its database is virtualized. If the monitoring tool is not virtually aware or simply monitors at the guest OS level, then there can be CPU clock drift and skew as seen by the monitoring software. This can lead to incorrect diagnostics and remedies. Furthermore, none of the currently available database benchmarking tools inherently handle this because their job is just to be the treadmill and simply stress the database. So my advice is to invest in a good monitoring solution that's fully virtualization aware.

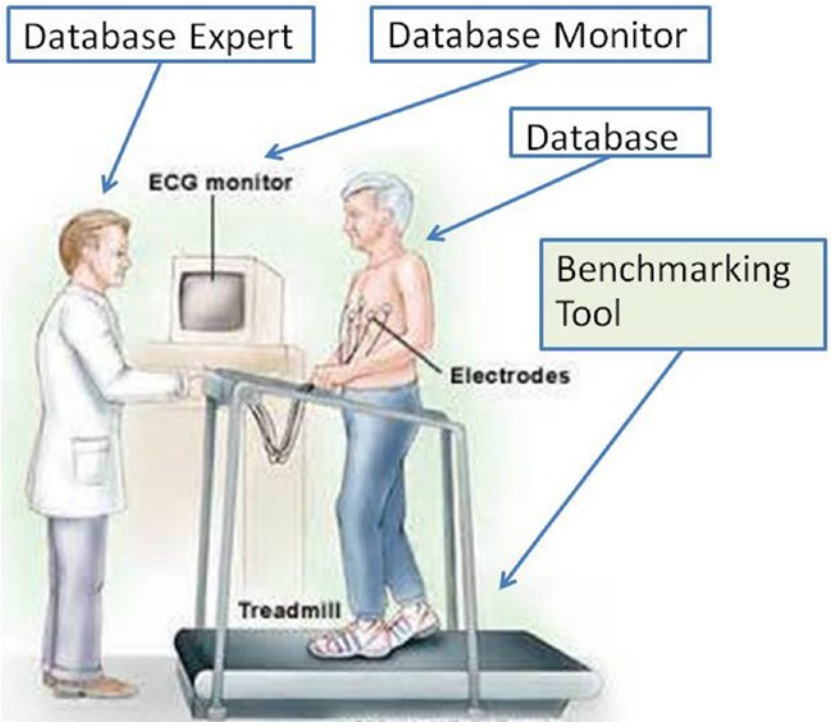


Figure 9-5. *Benchmarking like Cardiac Stress Test*

Summary

In this chapter we examined the issues and techniques for performing database benchmarking for virtualization. The basic message being that from the BIOS up through all the virtual machine hardware resource configuration settings, there are numerous choices to be made that affect achievable benchmarking performance. Therefore it's very worth the time to verify and, where necessary, correct these settings in order to meet the benchmarking expectations.

CHAPTER 10

Benchmarking for Public Cloud

In this chapter we're going to review benchmarking considerations for when you're considering or moving database instances and their databases to the public cloud. This chapter is purposefully kept short in order to focus on just the main issues, because the cloud has so many variables plus it is evolving every day. A whole book covering just the topic of deploying databases in the cloud would fill a volume all by itself. So here we must concentrate on the big-ticket items that are of the most value.

In this chapter the examples and quotes shown will be from Amazon's AWS and Microsoft Azure clouds. Again, I am not suggesting that you choose any particular vendor nor endorsing any vendor. But these two are the major players (for now). Finally note that the prior two chapters covering database consolidation and virtualization provide the basic foundation for deploying to the cloud. So if you have skipped those chapters or are not entirely comfortable with their concepts, it would be advisable to review them before processing.

Note While this chapter may highlight examples from a given public cloud vendor, all the principles and concepts shown in this chapter would apply for any cloud vendor's solution. The exact name or terminology might change, but the basic concepts would still apply.

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_10

Image Right Sizing

The first and most obvious thing DBAs need to define before deploying any database to the cloud is what type and size of image to select. The problem is that there are more choices than you can shake a stick at. Table 10-1 shows a simple breakdown by major category. Note that there are many different sizes per category, plus some vendors do not offer every category but rather allow users to add GPU's and extra storage to any configuration.

Table 10-1. *Various Major Cloud Image Categories*

	Amazon	Azure	Google
General Purpose	T2, M5, M4	B, Dsv3, Dv3, DSv2, Dv2, Av2	n1-standard
Compute Optimized	C5, C4	Fsv2, Fs, F	n1-highcpu
Memory Optimized	X1e, X1, R4	Esv3, Ev3, M, GS, G, DSv2, Dv2	n1-highmem, n1-ultramem
Storage Optimized	H1, I3, D2	Ls	<none>
Accelerated Computing	P3, P2, G3, F1	NV, NC, NCv2, NCv3, ND	<none>

Let's assume that your chosen cloud provider is Amazon, that you've read the details about their various image categories, and that you have selected image type M4. Now I am not promoting or hinting that this is the size to pick for a database. Here are the use cases recommended for M4: small and mid-size databases, data processing tasks that require additional memory, caching fleets, and for running back-end servers for SAP, Microsoft SharePoint, cluster computing, and other enterprise applications. The reason I chose M4 was simple: it has a reasonable number of sizes as shown in Table 10-2. If I had instead chosen M5, which might for many represent a good "*safe bet*" choice, the table of sizes would have spanned two pages or more. The point is that between choosing

the right image category and then the proper size for that category is a Herculean task. The good news is that you can change the size fairly easy. As for category changes, that depends on the cloud provider as to whether possible and, if so, how difficult.

Table 10-2. *Example Amazon M4 Image Sizes*

Model	vCPU	Mem (GiB)	SSD Storage (GB)	Dedicated EBS Bandwidth (Mbps)
m4.large	2	8	EBS-only	450
m4.xlarge	4	16	EBS-only	750
m4.2xlarge	8	32	EBS-only	1,000
m4.4xlarge	16	64	EBS-only	2,000
m4.10xlarge	40	160	EBS-only	4,000
m4.16xlarge	64	256	EBS-only	10,000

The important thing to keep in mind from a database benchmarking perspective is not to choose an artificially small size to save money or a large one to show optimum results. You should choose the same as you are planning to deploy for production on the cloud. So if you know, for example, that you're going to deploy Oracle 12g running under Oracle Enterprise Linux on an m5d.xlarge image (which includes NVMe SSD), then that's the size you should also chose for doing all your benchmarking. You may think this so obvious as to not need it said, but I have assisted many benchmarking efforts where clearly they fall into "*we're just doing this as a check-box item*" mode and so just use a small, cheap image. I'll just say that any such obtained results are arguably quite worthless. There are too many differences and options to base any conclusions when not choosing the same image category and size.

Deployment Architecture

The second and equally important thing DBAs need to define while deploying any database to the cloud is the deployment architecture. By that I mean where your images will be located. For that we first need to define a few terms specific to cloud deployment options: "*regions*" and "*availability zones.*"

A region is defined by dictionaries as an area or division, especially part of a country or the world, having definable characteristics but not always fixed boundaries. However, the simple, more colloquial meaning will generally suffice: a region is simply part of a country. For example, here in the United States we commonly think of the country as divided into five major regions: Northeast, Southeast, Midwest, Southwest, and West, as shown in Figure 10-1.

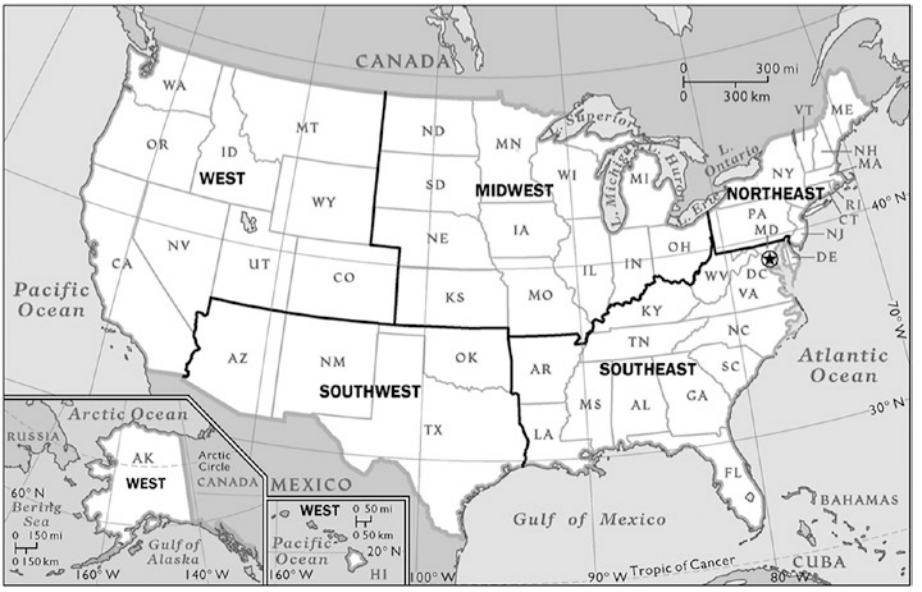


Figure 10-1. *Five Major Regions of the United States*

However, a region is defined by cloud providers as a set of datacenters deployed within a latency-defined perimeter and connected through a dedicated regional low-latency network. As one would expect, different cloud regions are geographically dispersed across major distances while its datacenters are more closely dispersed within that region. Table 10-3 shows the regions for Amazon and Azure.

Table 10-3. *Cloud Provider USA Regions*

Amazon		**Azure**	
Region	**Location**	**Region**	**Location**
us-east-1	US East (N. Virginia)	East US	Virginia
us-east-2	US East (Ohio)	East US 2	Virginia
us-west-1	US West (N. California)	Central US	Iowa
us-west-2	US West (Oregon)	North Central US	Illinois
		South Central US	Texas
		West Central US	Wyoming
		West US	California
		West US 2	Washington

An availability zone more or less equates to a datacenter within a given cloud region. Note that the datacenters are located far enough apart so that they should not be affected by common local problems such as a power outage or a weather storm, yet close enough to maintain low latencies between them. Some cloud providers are open about all their regions' datacenter locations, while others treat it as somewhat secretive. I guess that being "*in the cloud*" really means it's not that important as to the exact where as long as the regions' availability zones provide high resiliency across that region. Finally note that the number of availability zones varies by region. For example, on the Amazon cloud, the Virginia region offers six availability zones whereas Ohio has only three. That information could

be important, for example, if creating a clustered database and desiring a certain number of nodes in distinct availability zones. I generally prefer Amazon's diagram for these concepts, shown in Figure 10-2. Figure 10-3 shows the corresponding Azure diagram so that you can clearly appreciate that these concepts are basically identical across the various cloud providers.

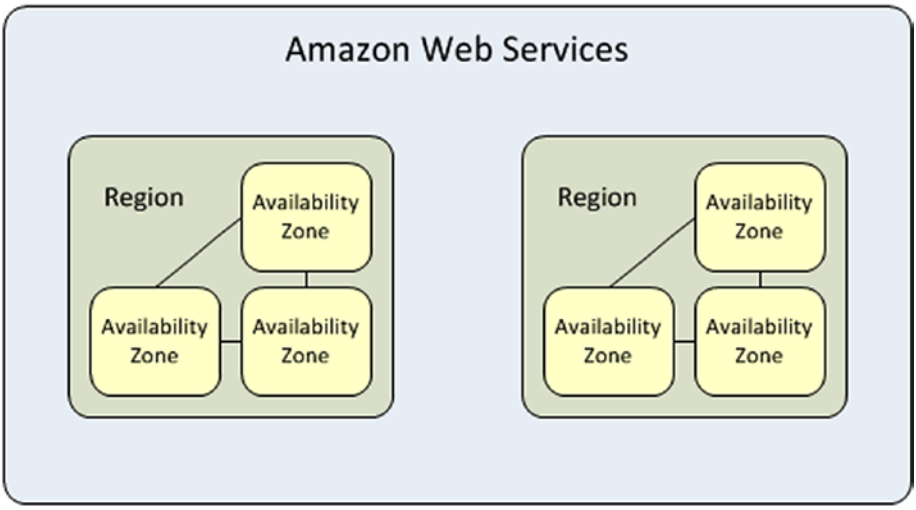


Figure 10-2. *Amazon Regions and Availability Zones*

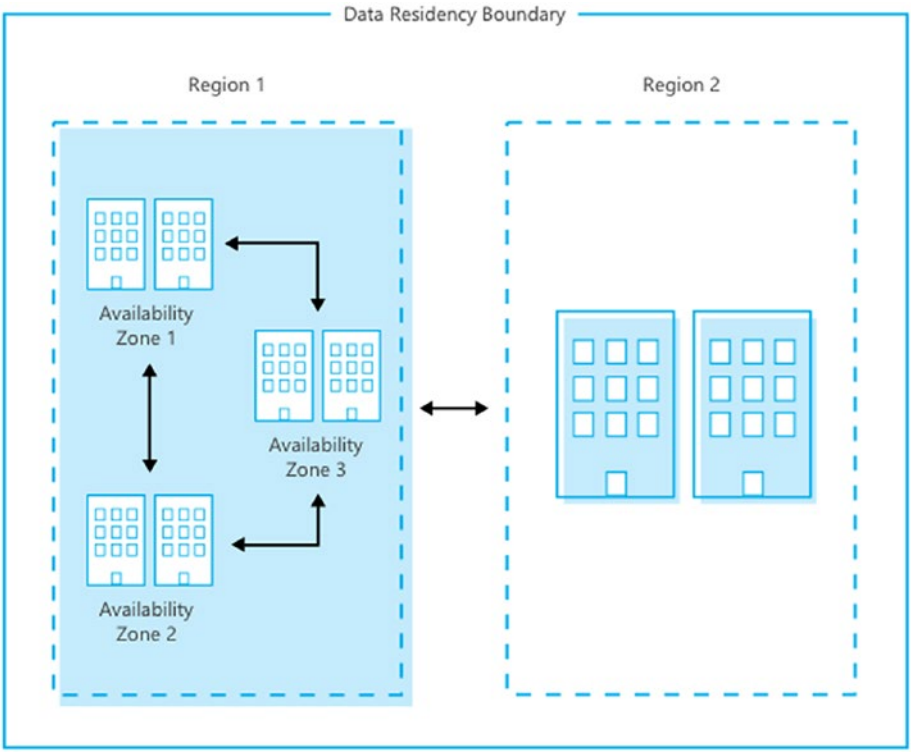


Figure 10-3. *Azure Regions and Availability Zones*

So how does all of this relate to database benchmarking? Simple, you should deploy your cloud images within the same region to avoid any latency skew. If working with a single instance (non-clustered) database, then you also might want to keep all your images within the same availability zone again to lessen any latency skew – even though latency between availability zones is kept low by default. It still is not as low as within the same availability zone. Now if you are working with a multi-node database instance where you might place one node in one availability zone and the other in a second, you might want to consider a split somewhat along the lines of that shown in Figure 10-4. This is essentially just a reworking of Figure 8-6 from Chapter 8. You should review that diagram and then examine how Figure 10-4 makes adjustments for the cloud and a multi-node, clustered database.

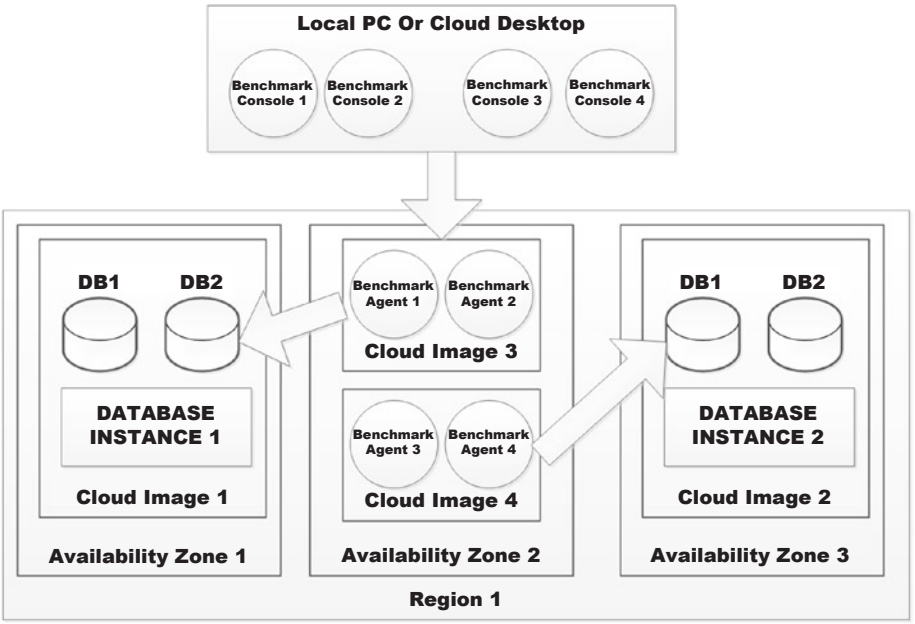


Figure 10-4. *Complex, Clustered Cloud Deployment*

Summary

In this chapter we examined the issues and techniques for performing database benchmarking for the public cloud. As was said, this chapter was kept short because one could write an entire book on cloud considerations for database deployment, tuning, and optimization. But two key concepts that must be both understood and properly selected are the image type and size, plus where the image will be deployed (i.e., regions and availability zones). We also stated that database benchmarking on a different image type, image size, or cloud image deployment model than what you expect to use for real is a near worthless exercise. You should always test with whatever you intend to use for real.

CHAPTER 11

Workload Capture and Replay

In this chapter we're going to review database workload capture and replay. This approach to stress testing or database benchmarking has some fairly obvious advantages. First, it's your application and database from which the transactional activity is derived. Therefore it has a direct correlation and relevance to your actual performance situation and issues at hand. Second, you already are intimately familiar with both the database design and the general business requirements being handled. So there are no specs to read and learn. Third, it's easier to research and propose potential solutions due to this familiarity. Plus it's easy to bounce ideas off of peers who also have such familiarity. For reasons such as these (and several others), some people might well consider this the ideal approach to stress testing or database benchmarking. It's hard to argue against such logic. I see their point. While Chapter 8 may have shown a way to create a mixture of industry-standard benchmarks in an attempt to approximate any system of interest, it nonetheless remains an approximation at best. We could easily forget some aspect that could radically skew the achievable results. You simply cannot be more accurate than running the actual workload.

There is yet still more good news regarding database workload capture and reply. While for a fair number of years there were solutions from third-party software vendors, in recent years many of the database vendors

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_11

have begun to offer such tools. For those who prefer workload capture and replay, this is a major victory as it's not uncommon to prefer a database vendor provided solution for certain key database management tasks where the vendor is perceived to have a natural advantage (e.g., database monitoring, database replication, database backup, etc.). Moreover, not only have the key database vendors entered this space, but they have rapidly evolved and improved their offerings. Therefore, these days DBAs have more and better options than in any time prior. This is very important as we're being asked to handle vastly different deployments such as databases on hypervisors, within containers, or in the cloud. Plus with advanced storage options such as SSD and NVMe in all those options, there are yet additional layers of complexity to overcome. Having the option for database workload capture and replay plays an important role in mastering all these situations.

Double Everything

The first and most obvious thing DBAs need to create in order to perform a successful workload replay is to make a copy of the source database and its original state of data. That copy will then be used as a separate target for the replay. Understandably you will need to take the copy before capturing the workload from the source such that you have a copy of the database at time zero (**T0**) before any transactions have been run that might change the data. That copy will then need to be loaded into the target such that any workload captured operating on **T0** data will then be able to be replayed with the exact same data and in the same exact order to hopefully produce the exact same results.

With the workload captured, there are two key replay scenarios. The first is to simply diagnose and solve a performance problem without any major charges to either hardware or software. This is probably the more common and likely scenario. This could include testing related to

minor operating system or database patches. The second is to test some significant change in hardware, software, or in the deployment model, such as virtualizing or moving to the cloud. While this scenario might not be as common, it nonetheless could be the most important in terms of strategy and long-term success. Let's look at both scenarios.

In the first scenario, we are testing simple performance optimizations without major platform changes. This means is that we need two database licenses, two servers, and double the disk space. Moreover we should strive to keep the servers and disk storage nature as similar as possible to the original. So if our production database server has dual octa core Xeons with 256GB of memory with 10TB of RAID-10 storage, then our test system for the replay should be the same or as similar as possible. Plus if our production database license is the enterprise edition with some optional cost add-ons, then we must also purchase the same for the test machine. The idea being that if mostly identical, then any tuning or optimization made on the target will most likely work on the source. That way we can find what things can be done to improve our production database.

Note, too, that we'll also need some disk space that can be mounted and shared with both machines to hold the workload capture files. These files could be quite large depending on the workload, so plan accordingly. You also will want to guarantee that the IO bandwidth to that shared space is sufficiently capable such that it does not skew either the source or target database servers. This is especially critical on the productions server since users must not be able to perceive that this effort caused any noticeable slowdowns that impact their normal work. No matter how you slice this, it means extra costs and risks.

In the second scenario, things are less constrained due to the very nature of what's being proposed and thus tested. We might be attempting to reduce database licensing fees by switching over to a "*standard edition*," for example. We might be comparing the current server vendor to another (e.g., Dell vs. HP vs. Lenovo). We might want to see if we can virtualize the database or move it to the cloud. We could just be looking at replacing

the current NAS or SAN device with either a new vendor or an entirely new technology such as SSD or VVMe to replace a spinning disk. So while we might require two of certain categories of each component, they very well could be quite different as that's the purpose of the test – to test that difference.

Once you have all this necessary infrastructure in place, then you can usually copy the entire database via a standard backup and recovery, and since you most likely already perform backups no overhead is introduced. You will just need to time your workload capture to start after a full backup such that you have true **T0** data for both the capture and replay. Once you have that you can restore the target system back to the initial state without any impact to the production system.

Capture/Replay Tools

There are essentially four types of tools for performing database workload capture and replay:

- Database benchmarking tools offering capture/relay using native database trace files or profiles (e.g., HammerDB, Swingbench combined with Trace Analyzer, and Quest Software's Benchmark Factory)
- Database vendor tools offering intrinsic and low-overhead capture/replay mechanisms (e.g., SQL Server Distributed Replay and Oracle Real Application Testing a.k.a. RAT)
- Third-party vendor database capture replay tools (e.g., ExactSolution's iReplay, and ITGain's SQLReplayer)
- Open source database capture/replay tools (e.g., Query Store Replay for SQL Server)

As this book is all about database benchmarking, we're going to restrict our discussion to benchmarking tools that offer workload capture/replay since the vendor solutions, while often very good, only work for their own database. Besides, for some database vendors, this feature is an extra cost item that must be purchased for both the source and target databases, so once again double the costs.

Capture Duration

The first and primary consideration that the DBA overseeing a workload capture must decide is the time and duration. This is critical. While it may seem like an obvious and simple issue, the most common mistake I've seen is not picking an appropriate workload capture period. It should be during an exemplary period of the day when the workload is high. Yes, this will add overhead to production, which is never a welcome thing. But to capture the workload during any other time yields far less-effective replays for proper tuning and optimization. In fact, I'd argue that if your need for a workload capture and replay does not justify performing it during a peak activity period, then it's probably not worth doing at all. Imagine a surgeon having to make a decision regarding an amputation due to infected tissue spreading. That's a no-brainer decision that would be done as soon as possible because the cost of not doing it is high. If management says this task is important yet they will not permit performing the capture during a peak period, then you should explain the fallacy in such a position and push hard to be allowed to do a one-time capture that has value regardless of cost. When the answer is still no, then you should argue against doing it at all or at least lower the expectations on the value and thus achievable results given a suboptimal capture.

Once you get approval for a peak activity period, the next question is how long to run the capture. We want to run the capture long enough to collect a meaningful and useful sample. But we also want to run it as

quick as possible to minimize impacts to production and end users. I've seen people perform a capture from as short as 15 minutes to as long as 8 hours (and once even 24 hours, but that was a rare exception). While there is no true right or wrong answer, I am going to suggest that those two extremes are most likely wrong. The best way to pick the duration is to ask the businesspeople what are the true mission-critical, peak workload hours. For example, while an example OLTP system might process activity primarily from 8 a.m. to 5 p.m., it may be that the business users can shorten that duration by identifying a key range within that period. Maybe because the business spans the four time zones in the United States that the key period is when all time zones are in normal business hours highlighted in Table 11-1. Therefore, the best period is anytime from 11 a.m. to 5 p.m. Since there is a traditional one-hour, noon lunch break for all employees, we strike out those rows that contain 12:00. Therefore we now have but two options: from 11:00 to 12:00 or from 4:00 to 5:00. It's not always this easy, but the idea is that by asking business users some questions that options for a good period will present themselves.

Table 11-1. *Finding the Optimal Capture Period*

ET	CT	MT	PT
8:00			
9:00	8:00		
10:00	9:00	8:00	
11:00	10:00	9:00	8:00
~~12:00~~	~~11:00~~	~~10:00~~	~~9:00~~
~~1:00~~	~~12:00~~	~~11:00~~	~~10:00~~
~~2:00~~	~~1:00~~	~~12:00~~	~~11:00~~
~~3:00~~	~~2:00~~	~~1:00~~	~~12:00~~
4:00	3:00	2:00	1:00
	4:00	3:00	2:00
		4:00	3:00
			4:00

I'll conclude this topic by stating that for me, the ideal capture period is often and typically no shorter than one hour and no longer than four hours. In general, you need to capture neither too little nor too much activity, but rather whatever is the right amount for the task at hand to be solved. There of course will be exceptions. For example, let's assume that quarter-end processing takes eight hours to perform its task, then that's the right period of time for a workload capture of that activity. A final consideration often will be the processing overhead and disk space required for the captured transactions. So while a longer period might be best, sometimes you may have to restrict to the minimum viable period due to these other considerations and their costs.

Replay Architecture

With this approach of using database benchmarking tools offering capture/relay, the deployment architecture for the benchmarking tool itself is critical. Look at Figure 11-1, and then ask yourself why such a complex picture?

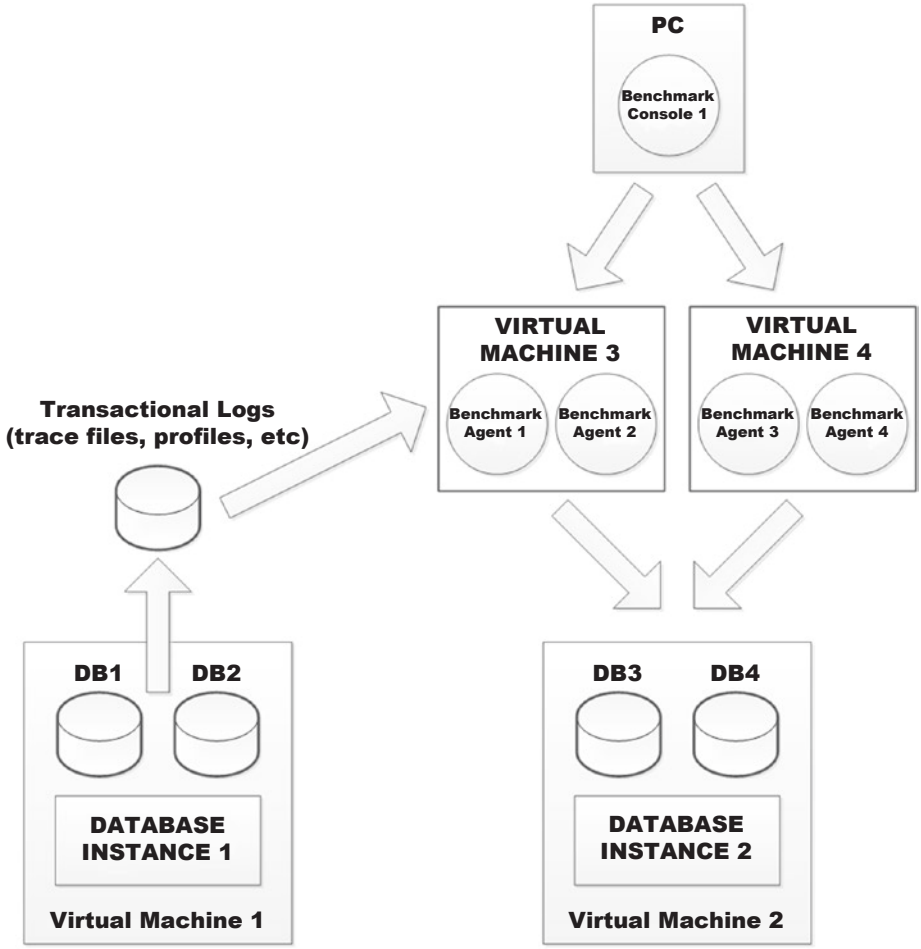


Figure 11-1. *Recommended Replay Architecture*

The answer is simple: you need sufficient benchmarking tool agents to adequately load balance all the work the database sees, which may have come from many application clients and servers. Imagine a traditional client/server type application being run by 10,000 users, or a web-based application server sending thousands of transactions per second to the database. You must have sufficient agents spread across computing resources to properly facsimilate the original workload delivery speed.

If not, the benchmarking agent could become a bottleneck that skews the results.

One final note, the database vendor capture/replay tools work nearly the same way in terms of the required architecture. While the database itself again captures the workload, you may need to deploy multiple database replay agents in order to load balance and approximate the delivery of the workload. Also note that for those databases that charge for the capability, they require you to license the capture/replay on both the source and target databases. In some cases this cost be quite substantial. But they do not charge extra for the agents required to perform the replay; it's just charged by the database instances.

Interpreting Results

For industry-standard database benchmarking scoring, the results are fairly easy; you can look at transactions per second, average response time, or runtime to completion. In other words, there are simple scores that one can compare that either improve or not between executions to test various database alterations. But it's not that easy with capture/replay efforts. Very few applications are properly metered in such a way as to provide simple performance comparisons. Plus not all databases inherently offer metrics such as transactions per second or megabytes of read/write per second. Plus even if they do, it's probable that the SQL code is the performance culprit more than anything else. Hence the best way to utilize replays is to use standard database monitoring tools to find and correct bad SQL. Moreover you might be able to identify SQL statements and their corresponding jobs that do not play well together. For example, at one company where I was the primary production DBA, the majority of our batch job cycles issues were simply a matter of rescheduling jobs that were resource intensive or needed to be hammered to the same tables.

Another method DBAs often like to pursue is to find key database wait events consuming too much time and correcting the root cause. Yes, sometimes it is a database configuration parameter or some other database issue. But most times the underlying cause is either bad SQL or SQL overlap that fight for competing database resources. So for those who prefer to start with wait events. I will not challenge your approach. I'll just say that 80% of the time or more. it will be SQL code in a production system that is the culprit. So why not look for such SQL first?

Capture/Replay Example

This section will demonstrate how to perform a workload capture replay using a database benchmarking tool. I am going to cheat though as I am going to also use the benchmarking tool to generate a workload for capture. So I'm going to run a TPC-H benchmark against the source database and capture that workload via the native database mechanism (in this case. Oracle trace files). Then I'm going to use the same benchmarking tool to read and replay those transactions. This example may not be very useful in itself, but it will fully demonstrate the proper process to follow. The benchmarking tool that we'll be using is Quest Software's Benchmark Factory.

The first step is to create two TPC-H database benchmarking jobs as recommended in Chapter 5: the first to load the database objects and the second to run the actual benchmark's workload. Second, run the database load object project. While the load project is running. we can make a minor change to execute the benchmark. Remember in this example we're working with Oracle, but the concept and process would be similar for other database platforms. We just need to manually add some statements to the benchmark's execution as shown in Figure 11-2. I've added

START_TRACE_1 and START_TRACE_2, which respectively contain the following two Oracle commands:

- ALTER SESSION SET tracefile_identifier=TPC_H;
- ALTER SESSION SET EVENTS'10046 trace name context forever, level 12';

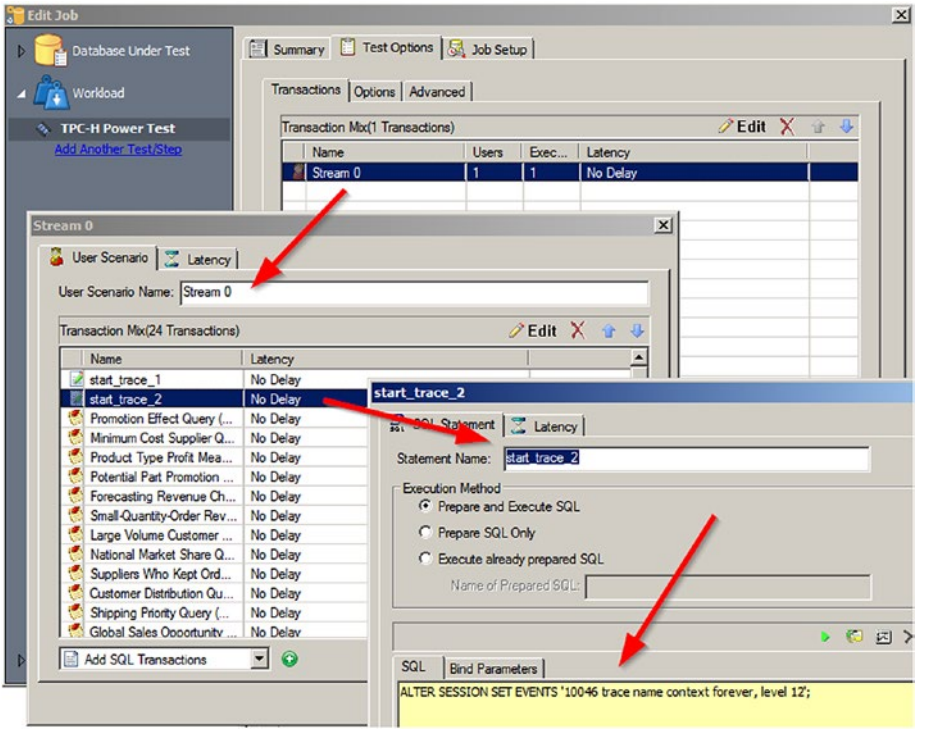


Figure 11-2. *Add Commands to Database Logs Workload*

By adding these two very special SQL statements, Oracle will now create trace files that contain as part of their name the string TPC_H to make those files more easily identifiable. Then I simply run the benchmark execution project that will run the TPC-H industry-standard benchmark's 22 complex SQL SELECT commands and generate a single trace file (containing all bind variable values) as shown in Figure 11-3.

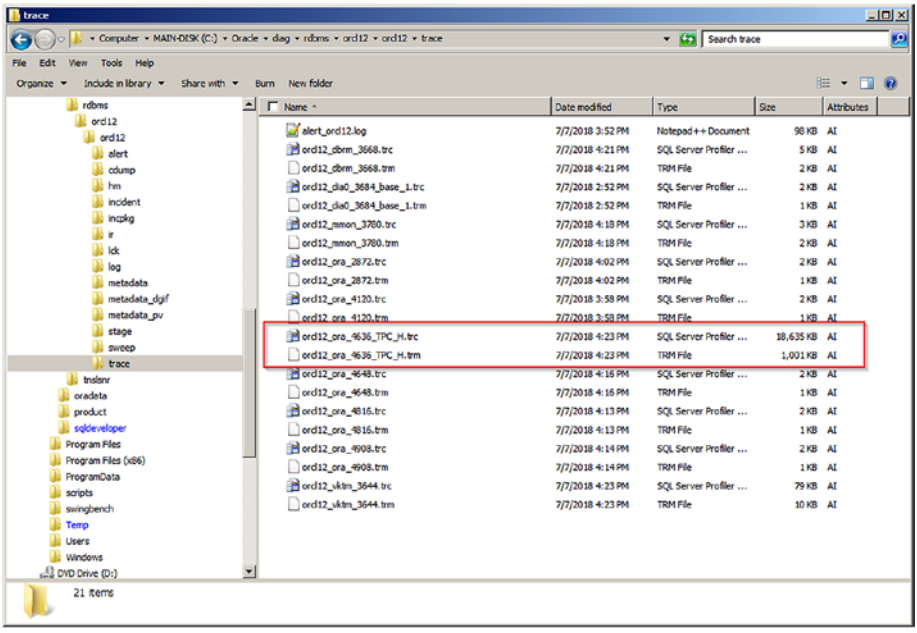


Figure 11-3. *Oracle Trace Files Created by Benchmark Run*

This trace file can then be loaded by the benchmarking tool as the commands for a project and run as shown in Figure 11-4. Note too in this figure that there are other options for creating a workload replay test other than just Oracle trace files. For example, if your application is metered and can be instructed to generate a text file log of SQL statements it runs and their bind values, then that can be parsed and loaded into this tool as well. So there are more than a few ways to perform workload capture and replays other than the database vendor solutions. While here I highlighted Benchmark Factory, both HammerDB and Swingbench offer similar features and capabilities.

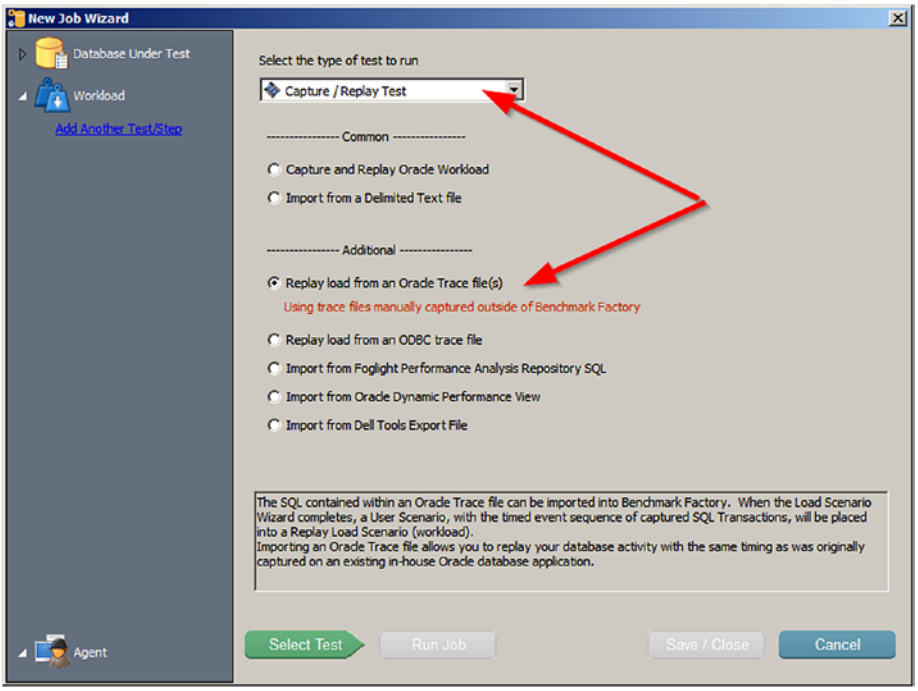


Figure 11-4. *Create Benchmark Job from a Trace File*

What's nice is that as Benchmark Factory loads and parses the trace file, it offers one the option to either import the entire file or to restrict it to some date/time range as shown in Figure 11-5. That's very useful since one might capture several hours' activity but only wish to replay a portion of the transactions. In fact, that's very useful when tuning and optimizing. Imagine running the workload replay the first time for 10:00 a.m. to 2:00 p.m. and being able to identify that the biggest problem occurs from 12:00 to 12:30. Now you might try to correct that issue by changing a database configuration parameter. Now you can simply rerun the workload replay and simply restrict it to just the time in question. This one capability alone is a great time saver that makes this feature worthwhile.

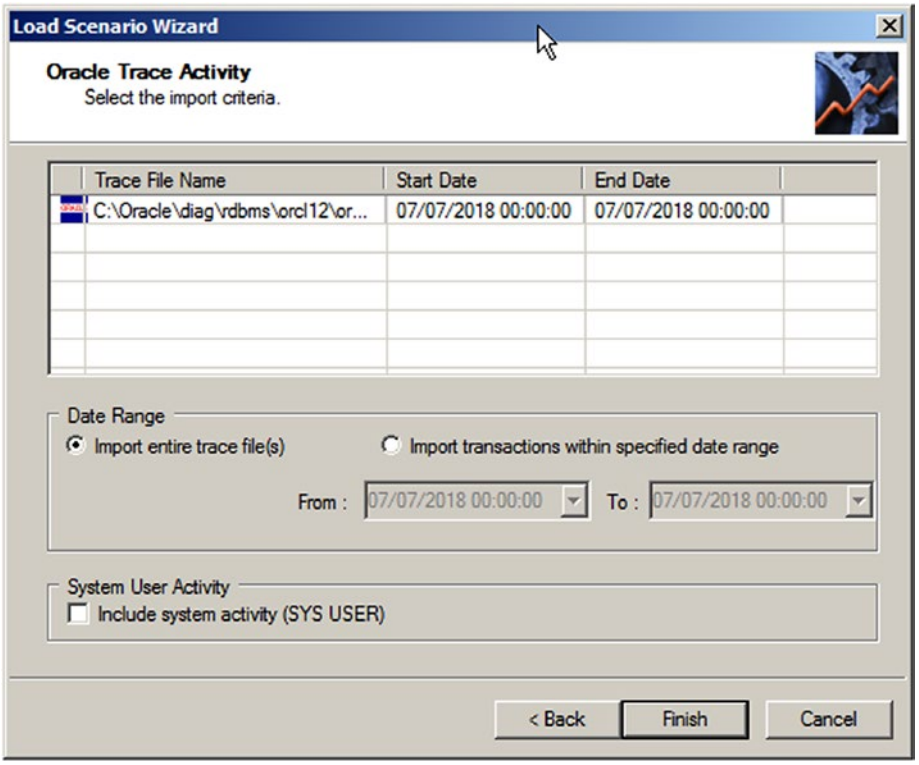


Figure 11-5. *Limit Trace File Input for Replay*

Summary

In this chapter we examined the issues and techniques for performing database workload capture and reply. The key issues were that workload capture/replay, by its very nature, requires double the hardware and software licenses. Plus you must identify the proper time period and duration for capturing a useful workload that's neither too brief nor too long. Then you need a capture/replay tool that works for your database platform, possibly from the vendor or from a third party. Then you can repetitively replay captured workloads in order to identify and correct SQL statement performance issues via whatever standard database monitoring and tuning tools you currently use. The goal is to simply improve the user's perception of the database system performance.

CHAPTER 12

Database Stress Testing

In this final chapter, we're going to discuss simple database stress testing, plus present a free scripting tool that I've written for people to utilize or alter to fit their specific needs. A stress test is much like an industry-standard benchmark in that it attempts to press the database. It's quite different as it applies simple, atomic transactions to merely stress the underlying subsystems, especially the IO bandwidth. I think of stress testing as being more like "*gunning the engine*" of an automobile, where the car is in neutral and you just want to see how high the RPM's can go before redlining. That does not mean that such testing is any less valuable, just that it basically serves a different testing purpose. For many people this is sufficient for what they might call database benchmarking.

Raw Stress Testing

As was discussed in the early chapters of this book, DBAs often just want to test raw database performance, which typically means to measure and compare some extremely low-level metrics such as IO's per second (IOPS). There are several reasons for this. First, IOPS is simple to understand and does not include any skewing factors like industry-standard benchmark delays. Second, storage vendors often express performance and throughput

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3_12

in these terms. Thus, assuming you want to measure "*raw performance*" metrics like IOPS, you'll want to use a database benchmarking tool whose design and purpose match that goal. One of the most popular lower-level database performance testing tools is Simple Little Oracle Benchmark (SLOB), which was mentioned near the end of Chapter 3. Many people swear by this tool, and several have written extensions or wrappers to improve upon it. But SLOB has notable limitations in my opinion:

- First, the logic is hard-coded in a C program and also depends on a C program for semaphore management. I prefer to keep all the code in shell script and ANSI SQL. which are all far easier for anyone to change as needed.
- Second, setup and running requires a multi-step process. I prefer one simple command interface with zero setup that you can simply invoke. Nothing to compile or link.
- Third, SLOB's core IO design is "reads" do lightweight block access and "writes" bypass index overhead. I prefer more natural database IO patterns more likely to match real-world database usage.
- Finally, the resulting SLOB load profiles are textual and for a single run (hence why some of the add-on wrappers and extensions).

I prefer a simple tool that does iterative testing and produces on one easy-to-read chart. Thus I wrote and have made freely available my own tool called DBBENCHMARK. It's basically a SLOB-like tool that completely addresses all these limitations. Furthermore, DBBENCHMARK is a simple Linux shell script using very few shells scripting "*special magic*," and so it can easily be converted to Power Shell for those wanting to run it on

Windows. There are very few Linux package dependencies, and it simply makes calls to the any database's command-line interface. So once I originally wrote this tool for Oracle, to convert it to MySQL was fairly trivial. Readers are welcome to convert it for PostgreSQL, SQL Server, or any other database. I merely request that people make such modifications publicly available for everyone's benefit.

Note You can download this tool from my personal website: www.bertscalzo.com.

DBBENCHMARK Tool

Running DBBENCHMARK is very easy – it's just a simple call to one Linux shell script requiring the user to specify just a couple relatively self-explanatory parameters as shown here for the Oracle version of the script:

```
[oracle@linux68 ~]$ ./dbbenchmark-oracle.sh -h

================================================================
Usage: dbbenchmark-oracle.sh -h -u -p -d -z -s -S -i -r -P -T -a -b
================================================================

CREDITS: utility for testing database performance by Bert Scalzo

OPTIONS:
   -h   Help          (print this help information)   --DEFAULTS--
   -u   Username       Database username                 ()
   -p   Password       Database password                 ()
   -d   Database       Database db name                  ()
   -z   Test Size      Values: SMALL, MEDIUM, LARGE      (SMALL)
```

```
   -s   Session Start  Beginning user session count    (1)
   -S   Session Stop   Ending user session count       (10)
   -i   Increment by   Increment session count by      (1)
   -r   Run Time       Run Time in Seconds             (30)
   -P   Plot Data      Plot graphs of SAR data: Y/N    (Y)
   -T   Tablespace     Default tablespace              (USERS)
   -a   AWR Snapshot   Take AWR snapshots: Y/N         (N)
   -b   AWR baseline   Create AWR baseline: Y/N        (N)

================================================================
```

Here is the MySQL version of the same script. Note of course that some parameters are either different or missing (i.e., MySQL does not have AWR facility). Remember too that this version calls a different command-line interface and passes some slightly different SQL to calculate the test results.

```
[oracle@linux68 ~]$ ./dbbenchmark-mysql.sh -h

================================================================
Usage: dbbenchmark-mysql.sh -h -u -p -d -z -s -S -i -r -P
================================================================

CREDITS: utility for testing database performance by Bert Scalzo

OPTIONS:
   -h    Help           (print this help information)   --DEFAULTS--
   -u    Username       Database username                  ()
   -p    Password       Database password                  ()
   -d    Database       Database db name                   ()
   -z    Test Size      Values: SMALL, MEDIUM, LARGE       (SMALL)
   -s    Session Start  Beginning user session count       (1)
```

```
   -S    Session Stop   Ending user session count       (10)
   -i    Increment by   Increment session count by      (1)
   -r    Run Time       Run Time in Seconds             (30)
   -P    Plot Data      Plot graphs of SAR data: Y/N    (Y)

=================================================================
```

Below is an actual example of running this utility against an Oracle 12c R2 database sitting on an SSD drive. I've specified only the minimum required parameters for the user name, password, and database. I also show only the beginning and ending portions of the script output, and not all 10 iterations (for 1 user to 10 users).

```
[oracle@linux68 ~]$ ./dbbenchmark-oracle.sh -u bert -p bert -d
ora122

=================================================================
PARAMETERS:
    DB_USERNAME    = bert
    DB_PASSWORD    = bert
    DB_DATABASE    = ora122
    TEST_SIZE      = SMALL (10,000 rows / session)
    SESSION_START  = 1
    SESSION_STOP   = 10
    SESSION_INCR   = 1
    RUN_TIME       = 30 (seconds)
    PLOT_DATA      = Y
    DEF_TSP        = USERS
    AWR_SNAP       = N
    AWR_BASE       = N
=================================================================

WORKING: Testing for gnuplot found executable in current $PATH
WORKING: Testing for gnuplot must minimally be version >= 4.2
```

```
WORKING: Testing for sqlplus found executable in current $PATH
WORKING: Testing connect to database with supplied parameters
WORKING: Processing steps for running benchmark test
WORKING: ....Create DBBENCHMARK_RESULTS performance measurement
         table
WORKING: Executing 1 sessions against DBBENCHMARK_TEST table
WORKING: ....Waiting on 1 sessions against DBBENCHMARK_TEST
         table
WORKING: DBBENCHMARK_RESULTS performance measurement stop time
...
WORKING: Executing 10 sessions against DBBENCHMARK_TEST table
WORKING: ....Waiting on 10 sessions against DBBENCHMARK_TEST
         table
WORKING: DBBENCHMARK_RESULTS performance measurement stop time

================================================================

RESULTS:

DBBENCHMARK_2018.07.13_14.18.37/DBBENCHMARK__SAR_AVERGAGE_IO.log
#users          read_tps        write_tps
1               0.00            3.78
2               0.00            3.69
3               0.10            3.98
4               0.00            3.78
5               0.00            3.27
6               0.00            3.39
7               0.07            4.32
8               0.14            4.54
9               0.07            5.69
10              0.00            4.09

DBBENCHMARK_2018.07.13_14.18.37/DBBENCHMARK__SAR_AVERGAGE_CPU.log
```

```
#users      %user       %nice       %system     %iowait
1           0.39        0.00        0.48        0.28
2           0.33        0.00        0.43        0.29
3           0.33        0.00        0.47        0.30
4           0.32        0.00        0.49        0.29
5           0.34        0.00        0.44        0.29
6           0.52        0.00        0.56        0.30
7           0.39        0.00        0.47        0.31
8           2.83        0.00        1.23        0.28
9           0.65        0.00        0.57        0.29
10          1.11        0.00        0.69        0.29

DBBENCHMARK_2018.07.13_14.18.37/DBBENCHMARK__SAR_AVERGAGE_RAM.log
#users      %mem_used
1           79.40
2           79.40
3           79.42
4           79.42
5           79.39
6           79.40
7           79.42
8           79.45
9           79.46
10          79.44

DBBENCHMARK_2018.07.13_14.18.37/DBBENCHMARK__SAR_AVERGAGE_QUE.log
#users      run_queue
1           1
2           1
3           3
4           1
5           0
6           2
```

```
7               1
8               0
9               0
10              2

================================================================

WORKING: Zipping up test report directory and files into one
zip file

================================================================

ALLDONE: Processing successfully completed ...

================================================================
```

DBBENCHMARK will create both a directory and a single zip file of that directory's content, both using the following naming format:

DBBENCHMARK_ + YYYY_MM_DD_ + HH.MM.SS

There will be three types of files in that directory: SAR files per iteration, rollups of the key SAR data, and four GNU Plot JPEG files. Figure 12-1 shows an example of this directory structure and file names.

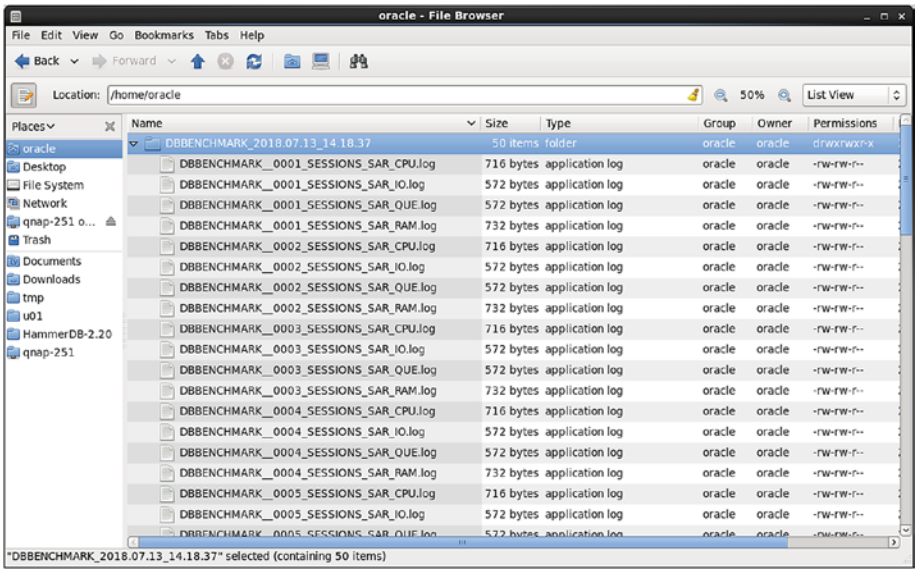


Figure 12-1. *Example Output from DBBENCHMARK Run*

These log files contain lots of detailed monitoring information that is summarized by iteration for each of four major categories (CPU, RAM, IO, and run queues) at the end of utility's text output as shown above. But if you're anything like me, you really require a simple picture to more fully comprehend the true findings in the results. Thus, DBBENCHMARK can automatically actually plot all the key results for you in the JPEG files in this directory. The first JPEG file, shown in Figure 12-2, shows the CPU usage. It's clear that from a processor perspective, this database server can probably handle well over 100 concurrent users. CPU is not the bottleneck.

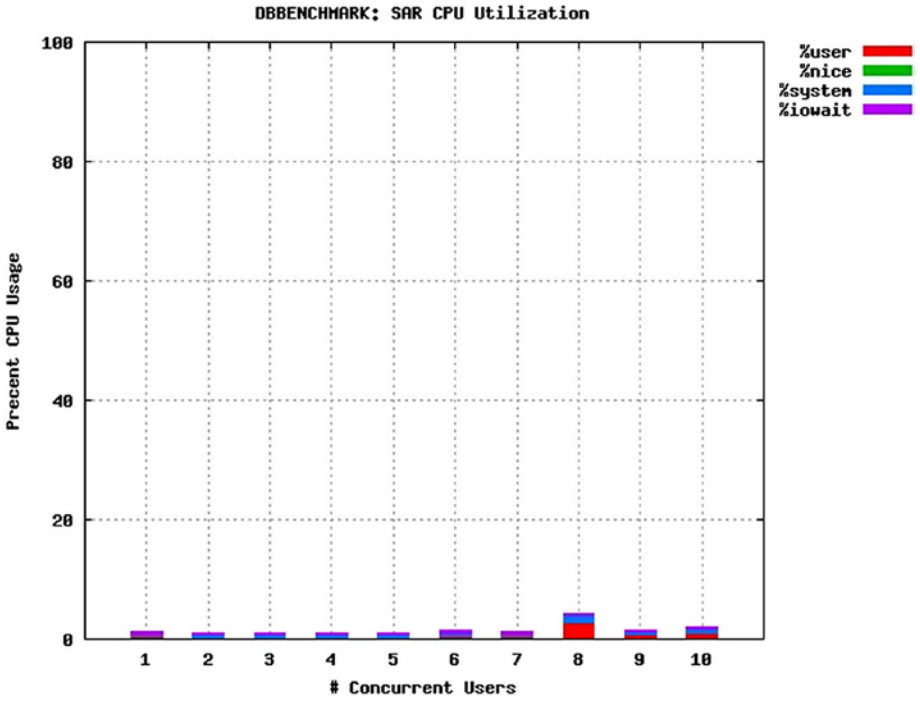


Figure 12-2. *DBBENCMARK Generated CPU Diagram*

The second JPEG file, shown in Figure 12-3, shows the average run queue length – often a key metric for detecting an overloaded system. It's clear that from a run queue perspective, this database server has room left. You often can scale up until the run queue depth averages consistently above four. So the run queue is not the bottleneck either.

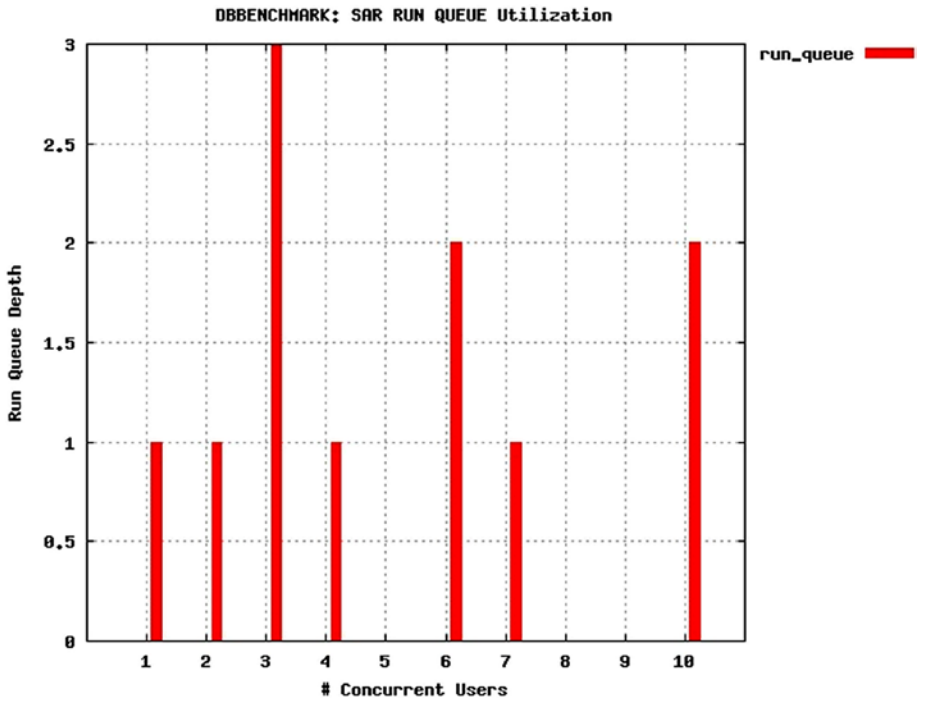


Figure 12-3. *DBBENCMARK Generated RUN QUE Diagram*

The third JPEG file, shown in Figure 12-4, shows the RAM usage. It's clear that from a memory perspective, this database server is running right around the ideal 80% memory level, so it may well be a limiting factor far more than CPU. While I might not be able to allocate any additional memory to the database for caching, there seems to be sufficient memory for spawning OS sessions to run the command-line interface. But it's probably true that memory will constrain the upper limit far more than CPU. So the memory is probably not the bottleneck either.

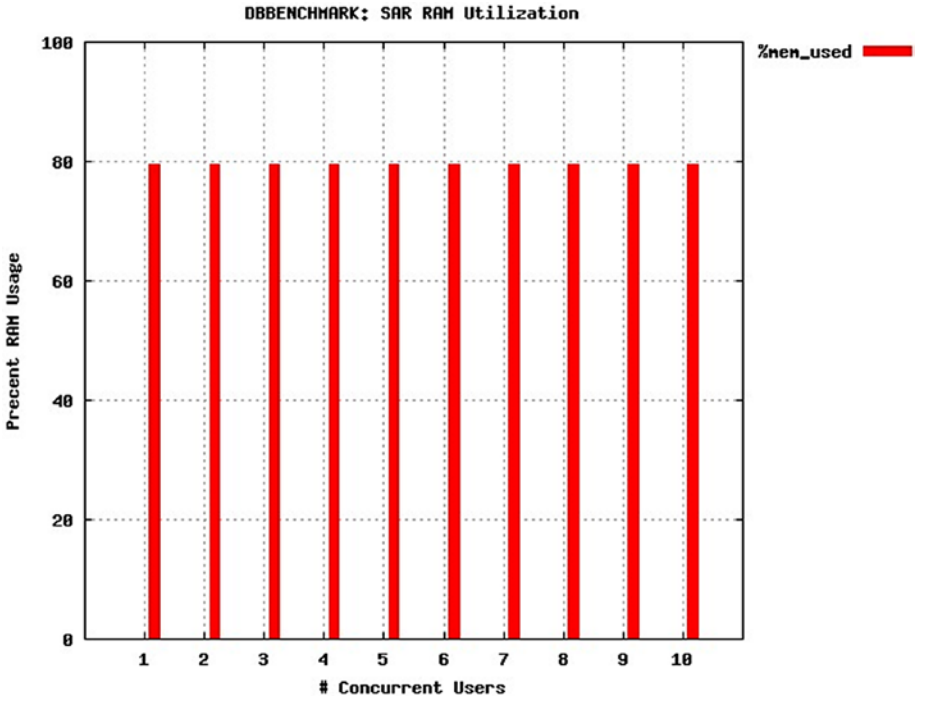


Figure 12-4. *DBBENCMARK Generated RAM Diagram*

The fourth JPEG file, shown in Figure 12-5, shows the read and write transaction rates. It's clear that from an IO perspective, this database server is again lightly loaded. An SSD disk should be able to handle lots more than six IO operations per second. So the IO also is not the bottleneck.

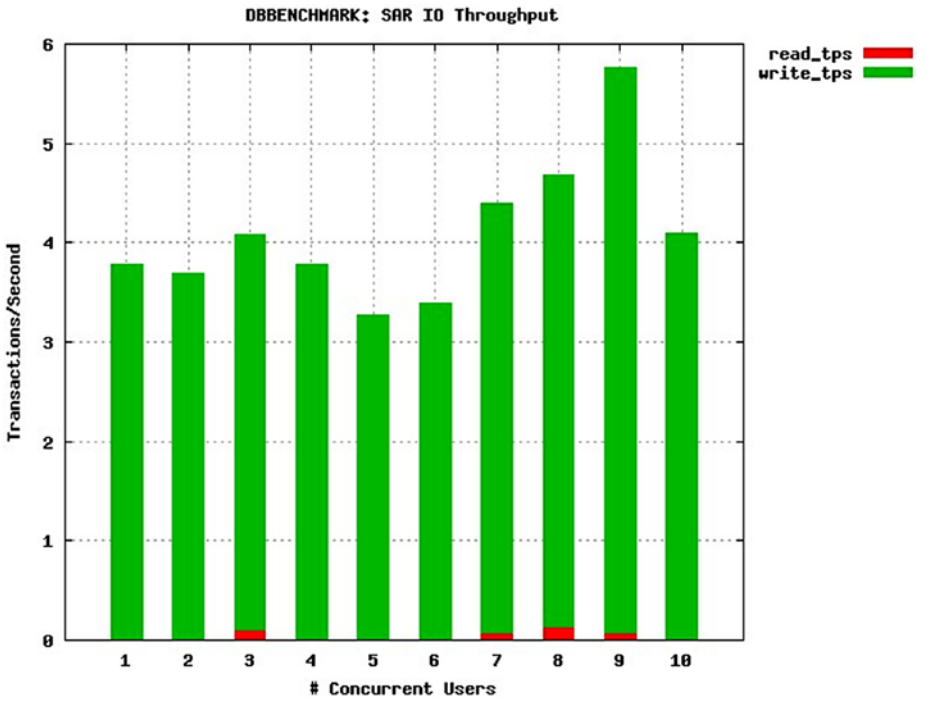


Figure 12-5. *DBBENCMARK Generated IO Diagram*

What these four diagrams tell me is that while I was afraid to run more than 10 concurrent user sessions on this notebook computer, that in fact it can handle far more workload. In fact, I did not reach a maximum until I increased the parameters for sessions from 1 to 10 increment by 1, to 20 to 200 increment by 10. That test range yielded that the greatest throughput or inflection point of diminishing results around 180 concurrent database sessions. While the IO and run queue lengths both maxed out – I was still under 20% CPU utilization. So even this modest notebook could scale even greater with additional SSD disks added to spread the IO load.

Summary

In this chapter I presented you with a free database stress testing tool known as DBBENCHMARK that's easy to use and does not require compiling any code like SLOB does. Plus you can easily edit the script to work for just about any database. I have provided both Oracle and MySQL versions, but to switch to PostgreSQL or just about any other relational database should be at most an hour's work. The real beauty behind DBBENCHMARK is that it can graph the results so that you don't have to look through lots of monitoring or logging output to find patterns. The pictures are truly worth a thousand words. I also showed that using DBBENCMARK that I was able to figure in very little time and with very little effort that my notebook computer with a single SSD can handle no more than 180 concurrent database sessions. I'd need to add more disks to spread the IO in order to scale higher. And it took less than an hour to find the magic number. Your mileage will vary, but you should be able to just as easily figure the max workload for just about any database server with this tool.

Index

© Bert Scalzo 2018
B. Scalzo, *Database Benchmarking and Stress Testing*,
https://doi.org/10.1007/978-1-4842-4008-3

E, F, G

H

I, J, K

L

M

N

O

P

Q, R

S

T

U

V

W, X, Y, Z