Benchmarking with TPC-H on Off-the-Shelf Hardware: An Experiments Report

Conference Paper · January 2012

CITATIONS
4

READS
421

3 authors, including:

Paulo Jorge Carreira
Technical University of Lisbon
53 PUBLICATIONS **240** CITATIONS
SEE PROFILE

Helena Galhardas
University of Lisbon
54 PUBLICATIONS **865** CITATIONS
SEE PROFILE

Benchmarking with TPC-H on Off-the-Shelf Hardware
An Experiments Report

Anna Thanopoulou[1], Paulo Carreira[1,2], Helena Galhardas[1,2]
[1]*Department of Computer Science and Engineering, Technical University of Lisbon, Lisbon, Portugal*
[2]*INESC-ID, Lisbon, Portugal*
anna.thanopoulou@gmail.com, paulo.carreira@ist.utl.pt, helena.galhardas@ist.utl.pt

Keywords: Database Benchmarking, Database Performance Tuning, Decision Support

Abstract: Most medium-sized enterprises run their databases on inexpensive off-the-shelf hardware; still, answers to quite complex queries, like ad-hoc Decision Support System (DSS) ones, are required within a reasonable time window. Therefore, it becomes increasingly important that the chosen database system and its tuning be optimal for the specific database size and design. Such optimization could occur in-house, based on tests with academic database benchmarks adapted to the small-scale, easy-to-use requirements of a medium-sized enterprise. This paper focuses on industry standard TPC-H database benchmark that aims at measuring the performance of ad-hoc DSS queries. Since the only available TPC-H results feature large databases and run on high-end hardware, we attempt to assess whether the standard test is meaningfully downscalable and can be performed on off-the-shelf hardware, common in medium-sized enterprises. We present in detail the benchmark and the steps that a non-expert must take to run a benchmark test following the TPC-H specifications. In addition, we report our own benchmark tests, comparing an open-source and a commercial database server running on off-the-shelf inexpensive hardware under a number of equivalent configurations, varying parameters that affect the performance of DSS queries.

1 INTRODUCTION

In order to keep track of their activity, enterprises need well-structured databases, where to record details about people (e.g., suppliers, clients, employees), items (e.g., raw materials, plants and equipment, inventory) and transactions (e.g., supplies or equipment purchases, products sales, expenses). In smaller enterprises, the database is hosted on inexpensive, off-the-shelf hardware and software administered using basic rules-of-thumb.

In the day-to-day operations of this enterprise, two types of queries are executed: the *Online Transaction Processing (OLTP)* and the *Decision Support (DSS)* ones. The former are basic information-retrieval and -renewal functions, such as looking up the address of a client or registering a new transaction. The latter are aimed at assisting management decisions regarding the company based on performance indicators and statistics retrieved from historical data. DSS queries tend to be far more complex, deal with a larger volume of data and thus take much longer to return results than OLTP ones. In addition, DSS queries can be further categorized into reporting and ad-hoc queries. Reporting DSS queries are routinely executed every now and then. For example, at the end of each month, management asks for the account with the higher performance in terms of sales volume. Conversely, ad-hoc DSS queries are executed on the spot when executives need a specific answer. For example, management may be considering applying some price discrimination technique according to customer order size. In order to verify the profitability of such action, they need to retrieve the distribution of customers by the size of orders they have made.

As one would expect, from a database administration point of view, the most challenging queries are the ad-hoc DSS ones. Not only are they highly complex, but they are also introduced in a spontaneous fashion, therefore not permitting the database designer to plan accordingly and optimize performance. Hence, it is very important to take all measures at our disposal to facilitate the execution of these queries, since they may end up running for hours or even days, and may impair the execution of the usual OLTP queries.

The time needed to execute ad-hoc DSS queries is above all related to the database design and size. Fur-

thermore, for a given database, time depends on the choice of DBMS and its tuning. Given the wide offer of database systems nowadays as well as great complexity, it is crucial yet not trivial for the enterprise to determine the best choice for its needs, both in terms of price and in terms of performance. The obvious answer is that one should choose the system achieving the required performance level at the minimum cost. Therefore, it would be helpful to realize a quantitative comparison of database systems performance under various comparable configurations, possibly using a benchmark. A *benchmark* is a standardized test that aims at comparing the performance of different systems under the same conditions, presenting results as work executed per time or money unit.

The Transaction Processing Performance Council (TPC) benchmark TPC-H constitutes an attempt to model a business database along with realistic ad-hoc DSS questions. It has been extensively used by database software and hardware vendors to demonstrate the performance of their solutions, as well as by researchers looking to validate their approaches. However, TPC-H official results refer to very large databases running on high-end hardware that are difficult to compare to the reality of a small enterprise. Moreover, it is not clear from the text of the standard how to run the tests and whether a simple procedure to run them exists enabling small enterprises *(i)* to analyze different off-the-shelf DBMS and their configurations and *(ii)* to evaluate different tuning options.

This paper will examine whether TPC-H can be used as a tool by small enterprises to facilitate their database administration decisions, and it will compare the performance of a commercial and an open-source database system under different tuning settings when tested at this scale. Specifically, our contributions include: *(i)* a detailed description of all the steps necessary to execute the TPC-H test on off-the-shelf hardware aimed at the non-expert; *(ii)* a comparison of the performance of a commercial and an open-source database system executing a small-scale TPC-H test under various comparable configurations on off-the-shelf hardware; *(iii)* insights into the tuning parameters that influence DSS performance at this scale; and *(iv)* an evaluation of the usefulness of the TPC-H test at lower scales.

Our text is organized as follows. In Section 2, we present some related work on the subject of optimizing DSS queries. Sections 3 and 4 offer a presentation of the main features of the TPC-H schema and workload, as well as the research methodology used during our TPC-H tests. Section 5 presents the results of our full TPC-H test results for various tunings of SQL Server 2008 and MySQL 5.1. Finally, in Section 6 we present our conclusions and outlook for future work.

2 RELATED WORK

The TPC-H databases and workload have been used extensively in research as a means of validating the superiority, in terms of performance of a breath of techniques related to multiple aspects of data storage, retrieval and processing. (Harizopoulos et al., 2005; Somogyi et al., 2009; Guehis et al., 2009; Kitsuregawa et al., 2008; Lim et al., 2009). In addition, the benchmark has been analyzed (Seng,), synthesized (**?**) and reduced to a handful representative queries (Vandierendonck and Trancoso, 2006). There even have been attempts to propose alternative benchmarks (O'Neil et al., 2009; Darmont et al., ; Siqueira et al., 2010), mainly aiming at proposing a star schema that favors data warehousing; an issue soon to be resolved by benchmark TPC-DS (Poess et al., 2002).

Understanding TPC-H requires significant technical expertise and, to the best of our knowledge, no comprehensive account of the deployment of the standard exists in literature. Apart from generic guidelines for benchmark execution (Oracle, 2006; Scalzo, 2007), no step-by-step guide has been created for the non-expert. Commercial solutions such as Benchmark Factory (Software, 2008) that are capable of running TPC-H validate the need for in-house deployment of this standart.

Besides deployment of the benchmark, another consideration is selecting a minimal set of meaningful tuning parameters to be varied in basic, low-scale experiments to determine a somewhat optimal tuning for DSS workloads. Indeed, there has been considerable amount of research on the topic of profiling automatically characterising a database workload as OLTP or DSS (Elnaffar, 2002; Elnaffar et al., 2002; Elnaffar et al.,). Determining the nature of each workload will assist the development of self-tuning database systems (Chaudhuri and Narasayya, 2007; Wiese et al., 2008), as each workload type requires a different approach to database system tuning. Some database vendors and hardware manufacturers provide recommendations for tuning according to the workload type (IBM, 2004; Green, 2002; Paulsell, 1999; Packer and Press, 2001). In the case of TPC-H DSS queries, disk I/O has been isolated as the limiting factor for performance (DeSota, 2001; Kandaswamy and Knighten, 2000). We based our recommendations for main tuning parameters to be varied in low-scale experiments with TPC-H on this conclusion.

3 AN OVERVIEW OF TPC-H

The IT department of a small enterprise can benefit from deploying a benchmark in-house in a number of ways. For example, to choose which database management system to use, given a set of hardware. In this case, the TPC-H test could be run on trial versions of various database systems against their machine and decide which one works best for them. There are numerous advantages of using TPC-H instead of the actual business database for this purpose. First, the TPC-H workload has been thoroughly prepared to include a number of representative examples of ad-hoc DSS queries with various levels of complexity. It would require a lot of technical expertise and time to "re-invent the wheel" within the company's IT department. Second, TPC-H allows the user to run experiments against small, medium, large or very large databases; hence, one does not necessarily need to run a test so long as it would have been against the actual database to arrive to meaningful conclusions. Third, using the company database in various systems to examine performance would involve re-writing code in a number of variations depending to each system's standard; TPC-H, conversely, allows the user to generate the test code for different systems in a straightforward way.

Other possible uses of TPC-H within the medium-sized company environment include choosing the hardware given the choice of a database system, deciding whether it is worthwhile in terms of price/performance to migrate to the new version of a database system, and locating whether a performance problem is due to system or database tuning.

3.1 The TPC-H Database Schema

The goal of the TPC-H benchmark is to portray the activity of a product supplying enterprise. Although the TPC-H standard only provides the relational schema, we present the corresponding E-R diagram in Figure 1, for clarity. The benchmark models any industry that manages, sells or distributes products worldwide, such as car rental, food distribution, parts or suppliers. The entity Part stands for an individual piece of product and the entity Supplier stands for a person or company that supplies the product for our corporation. The Order entity represents a single order, which has been placed by a customer represented by the Customer entity. Each order is constituted of lines of items represented by the Lineitem entity and are provided from a specific supplier's collection of parts. Finally, since both suppliers and customers are people, they are citizens of a particular Nation that belongs to a particular Region.

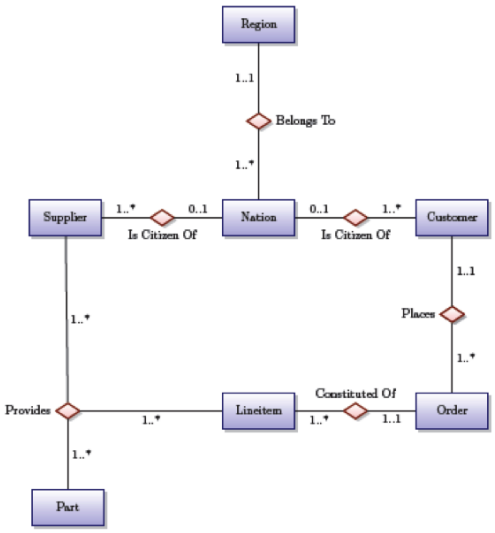

Figure 1: Simplified ER diagram of the TPC-H database model. Entity attributes are not displayed on the diagram.

This E-R diagram is converted into a relational schema comprised by eight base tables, as specified by TPC. Tables have different sizes that, except for nation and region, change proportionally to a constant known as *scale factor (SF)*, as seen in Table 1. The available scale factors are: 1, 10, 30, 100, 300, 1000, 3000, 10000, 30000 and 100000. The scale factor determines the size of the database in GB. For instance, choosing the scale factor to be equal to 1 means there will be generated 1GB of data in total, and so on.

Table 1: Number of rows per table in the TPC-H database relative to the scale factor. The two largest tables are Lineitem and Orders and hold about 83% of the total data.

Table Name	Number of Rows
Region	5
Nation	25
Supplier	$10,000 \times SF$
Customer	$150,000 \times SF$
Part	$200,000 \times SF$
Part_Supp	$300,000 \times SF$
Orders	$1,500,000 \times SF$
Lineitem	$6,000,000 \times SF$

3.1.1 Data Generation Using DBGEN

DBGEN is a data generator provided in the TPC-H package to fill the database tables with different amounts of synthetic data. The generation of synthetic data follows different rules according to the type of column. DBGEN uses grammar rules for each type of column to generate large amounts of synthetic data from a bank of base data elements within the

TPC-H package. Grammar rules enable guaranteeing that the synthetic data thus generated satisfies statistical properties such as having values distributed according to zipfian power law (Gray et al., 1994). As it will be made clear later, DBGEN is also used to produce random data for consequent line insertions to already populated tables.

3.2 The TPC-H Workload

The benchmark workload consists of 22 queries, representing frequently-asked decision-making questions, and 2 update procedures, representing periodic data refreshments. The update procedures are called *refresh functions* in the TPC-H specification document and we will refer to them as such in the rest of this document. The 22 queries implement mostly complex, distinct, analytical scenarios regarding such business areas such as pricing and promotion, supply and demand management, profit and revenue management, customer satisfaction survey, market share survey and shipping management. From a technical standpoint, the queries include a rich breadth of operators and selectivity constraints, access a large percentage of the populated data and tables and generate intensive disk and CPU activity. During the TPC-H test, each query will run *(i)* stand-alone to demonstrate the ability of the test system to use all of the resources for a single user, and then *(ii)* concurrently to assess the ability of the system to use all of the resources to satisfy concurrent workload.

3.2.1 Query Generation Using QGEN

The TPC-H workload queries are defined only as query templates in the specification document. That is to say, there is a functional query definition provided by TPC, defining in SQL-92 the task to be performed by the query. However, this definition is not complete; some parameters have to be instantiated (such as the name of the country for which we are going to scan table Nation in this run) in order to obtain a running query. The substitution parameters are generated by the application QGEN in such a way that the performance for a query with different substitution values is comparable. QGEN uses a data set of appropriate data types to fill in the query gaps, in a way similar to DBGEN. After running QGEN, we get ready-to-run queries in valid SQL, which have the same basic structure as the query templates but varying values for the substitution parameters. In order to validate the results of a query, the TPC-H specification includes validation output data for each of the queries for a specific value of the substitution parameters upon a 1GB database. We assume the correctness of QGEN in this respect. Note that QGEN creates only ready-to-run queries and not refresh functions. We can create our code for the latter ones, as explored in the following section.

3.3 TPC-H Tests

TPC-H comprises two tests: the load test and the performance test. The former involves loading the database with data and preparing for running the queries. The latter involves measuring the system's performance against a specific workload. We will discuss the exact steps that need to be taken and the values to be measured.

3.3.1 The Load Test

Preparatory steps leading to the load test consist of two phases: *(i)* creating the database, *(ii)* generating flat data files to populate it using the DBGEN tool provided by the TPC; then, we can go ahead and execute the load test. That is to say, we have to create the schema as specified by the TPC, load the data from the data files produced by DBGEN into the tables, add constraints (primary keys, foreign keys and check constraints) following the restrictions set in the TPC-H specification document, create the indexes, calculate the statistics for these indexes and install the refresh functions as stored procedures. These steps are illustrated in Figure 2. Although not crucial for DSS queries, the database load time is an important result to be reported, as it pictures the database system efficiency in setting up and populating a database—an operation that occurs at least each time new hardware is purchased and involves large amounts of data.

3.3.2 The Performance Test

As soon as the load test is complete, the performance test can start, which consists of two *runs*. Each run is an execution of the *power test* followed by an execution of the *throughput test*. In order to define the terms of power and throughput tests, we need to introduce the concept of *query* and *refresh streams*. A *query stream* is a sequential execution of each of the 22 TPC-H queries, while a *refresh stream* is a sequential execution of a number of pairs of refresh functions. Tests will consist of query streams and refresh streams. Figure 2 illustrates the steps for running a complete sequence of the TPC-H test.

3.3.3 Power Tests

The *power test* aims at measuring the raw query execution power of the system with a single active

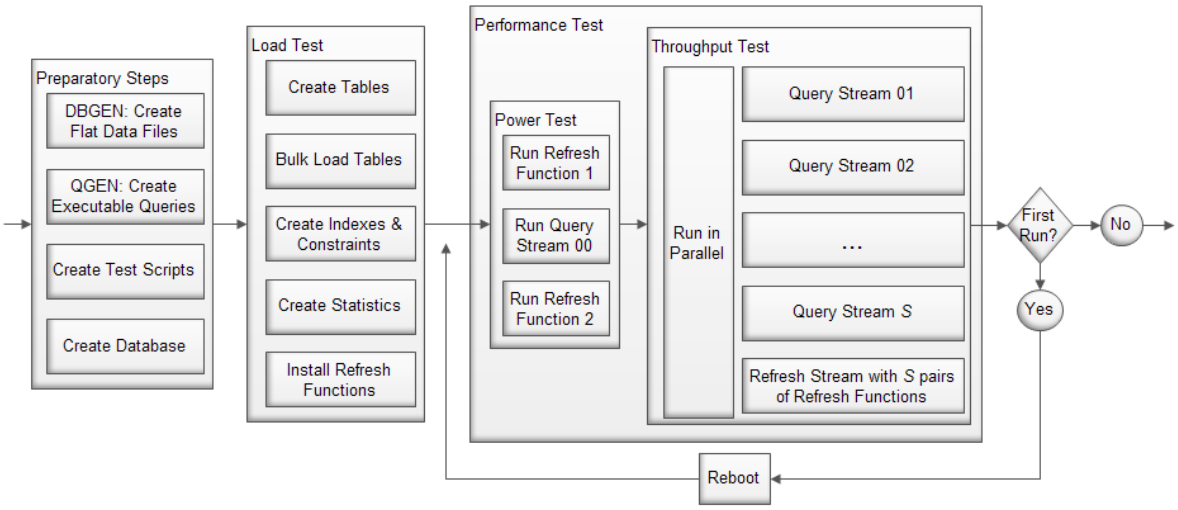

Figure 2: Complete process for running the TPC-H tests. From left to right, first we present the off-the-clock preparatory steps required to set up the test system including writing the required code, then we show the sequence in which each stream must run during the load, power and throughput tests. Note that the performance test is made up by the power and the throughput test, and it needs to be executed twice.

stream, that is to say how fast can the system compute the answer to single queries. This is achieved by running a single query stream, that is to say by sequentially running each one of the 22 queries. The power test also includes running a refresh stream session comprising a single pair of refresh functions.

In particular, there are three sequential steps necessary to implement the power test: *(i)* execution of the refresh function 1, *(ii)* execution of the query stream and *(iii)* execution of the refresh function 2. The query stream executed during the power test is called query stream 00. Correspondigly, the executed refresh stream is called refresh stream 00. The TPC specifies the exact execution sequence for the queries in the query stream 00, so that results do not vary according to the content of the cache memory based on previously-run queries.

3.3.4 Throughput Tests

The purpose of the *throughput test* is to measure the ability of the system to process the most queries in the least amount of time, possibly taking advantage of I/O and CPU parallelism. In other words, this test is used to demonstrate the performance of the system against a multi-user workload. For that reason, the throughput test includes at least two query stream sessions. Each stream executes queries serially but the streams themselves are executed in parallel.

The minimum number of query streams, referred to as **S** and specified by the TPC, increases with the increase of the scale factor, as shown in Table 2. What is more, the throughput test must be executed in parallel with a single refresh stream comprising *S* refresh function pairs. Each query stream and refresh function pair in the throughput test has an ordering number represented as **s** and ranging from 01 to *S*.

Table 2: The different scale factors (SF) along with the corresponding number of query streams (S) used in the throughput tests.

Scale Factor (SF)	Number of Query Streams (S)
1	2
10	3
30	4
100	5
300	6
1000	7
3000	8
10000	9
30000	10
100000	11

Like in the power test, the execution sequence for the queries in a query stream is pre-defined by TPC and determined by by each query's ordering number *s*. The purpose of this is to ensure that the different query streams running in parallel will not be executing the same query at the same time.

3.4 Performance Metrics

While running the power and the performance tests, the scripts will report the time for each one of the steps. Specifically, in the end we get three types of timing measurements regarding: the *database load time* as discussed in Section 3.3, the *measurement in-*

terval and the *timing intervals*. The measurement interval represented as T_s is the total time needed to execute the throughput test. The timing interval represented as $QI(i,s)$ is the execution time for the query Q_i within the query stream s, where i is the ordering number of the query ranging from 1 to 22 and s is 0 for the power test and the ordering number of the query stream for the throughput test. The timing interval represented as $RI(j,s)$ is the execution time for the refresh function RF_j within a refresh stream s, where j is the ordering number of the refresh function ranging from 1 to 2 and s is 0 for the power test and the position of the pair of refresh functions in the stream for the throughput test. All the timing results must be measured in seconds, as specified by the TPC.

Timing measurement results must then be combined to produce global, comparable metrics. In order to avoid confusion, TPC-H uses only one primary performance metric indexed by the database size: the *composite query-per-hour performance metric* represented as $QphH@Size$, where *Size* represents the size of data in the test database as implied by the scale factor. For instance, we can have the metric QphH@1GB for comparing systems using 1GB databases. This metric weighs evenly the contribution of the single user power metric and the multi-user throughput metric. We are now going to present in detail each one of them.

3.4.1 The Processing Power Metric

For a given database size, the *processing power metric* represented as $Power@Size$ is computed using the reciprocal of the geometric mean of the execution times for each one of the queries and the refresh functions obtained during the power test, represented as $QI(i,0)$ and $RI(j,0)$ respectively. Please recall that query and refresh streams in the power test have as ordering number s that is equal to 0.

The geometric mean indicates the central tendency of a set of numbers that are to be multiplied together to produce an outcome. Unlike arithmetic mean, instead of adding the set of numbers and then dividing the sum by the count of elements n, the numbers are multiplied and then the nth root of the resulting product is taken. For instance, the geometric mean of three numbers, 1, $1/2$, $1/4$ is the cube root of their product ($1/8$), which is $1/2$; that is $\sqrt[3]{1 \times 1/2 \times 1/4} = 1/2$. The intuition behind the geometric mean is perhaps best understood in terms of geometry. The geometric mean of two numbers, a and b, is the length of one side of a square whose area is equal to the area of a rectangle with sides of lengths a and b. Similarly, the geometric mean of three numbers, a, b, and c, is the length of one side of a cube whose volume is the same as that of a cuboid with sides whose lengths are equal to the three given numbers.

Query execution times of TPC-H are by design substantially different from one another, i.e. significantly too long or too short. This fact would influence the arithmetic mean, therefore the geometric mean is preferable as it is more robust. To illustrate the concept, consider a set of three queries with elapsed times of 10, 12 and 500 seconds. The arithmetic mean would be 174 seconds while the geometric one is 39.15 seconds. The processing power metric is defined as:

$$Power@Size = \frac{3600}{\sqrt[24]{\prod_{i=1}^{22} QI(i,0) \times \prod_{j=1}^{2} RI(j,0)}} \times SF$$

The denominator computes the geometric mean of the timing intervals for the 22 queries and the 2 refresh functions, in a total of 24 factors. It represents the aggregated effort in seconds to process a request, that being a query or a refresh function. The numerator 3600 is the number of seconds in an hour. Therefore, the fraction expresses the number of queries executed per hour. This number is then multiplied by the scale factor SF implied by the database size to give us Power@Size, where size is the GB implied by the scale factor. The units of the Power@Size metric are queries-per-hour $\times$ SF. Finally, since the TPC-H metrics reported for a given system must represent a conservative evaluation of the system's level of performance, the reported processing power metric must be for the run with the lower composite query-per-hour metric.

3.4.2 The Throughput Power Metric

The *throughput power metric* represented as $Throughput@Size$ is computed as the ratio between the total number of queries (executed within all the query streams of the throughput test) and the length of the total time required to run the throughput test for s streams, T_s. In simpler words, this metric tells us how many queries were executed in the elapsed time. Let s be the number of streams corresponding to the scale factor SF. The throughput power metric is defined as:

$$Throughput@Size = \frac{S \times 22}{T_s} \times 3600 \times SF$$

The numerator $S \times 22$ is the total number of executed queries within all streams (S streams with 22 queries each) and the denominator T_s is the total time for the throughput test in seconds. Therefore, the fraction represents the number of queries executed per second. Multiplied by 3600 seconds, it gives the number of

queries executed per hour. This result is then multiplied by the scale factor yielding Throughput@Size, where size is the GB implied by the scale factor. The units are queries-per-hour $\times$ SF. As in the case of Power@Size, the reported processing power metric must be for the run with the lower composite query-per-hour metric.

3.4.3 The Composite Query-Per-Hour Performance Metric

The *composite query-per-hour performance metric* combines the values of the corresponding metrics Power@Size and Throughput@Size into a single metric. This metric is obtained from the geometric mean of the previous two metrics. The metric captures the overall performance level of the system, both for single-user mode and multi-user mode. The composite query-per-hour metric is defined as:

$$QphH@Size = \sqrt{Power@Size \times Throughput@Size}$$

3.4.4 The Price/Performance Metric

This last metric allows test implementers to make the final price/performance comparison between systems (especially in our case enabling to compare commercial and open-source solutions) and decide which system gives the highest performance for money. The *price/performance metric* represented as $Price-per-QphH@Size$ is the ratio of the total system price divided by the composite query-per-hour performance metric. This price/performance metric is defined as:

$$\text{Price-per-QphH@Size} = \frac{\$}{QphH@Size}$$

Where the symbol \$ stands for the total system price in the reported currency. The units are the usual currency units.

4 EXPERIMENTS

The goal of our experiments is to showcase a set of useful TPC-H tests that any small entreprise could perform in order to choose the database system and tuning configurations that offer optimal ad-hoc DSS performance in their system. In addition, we run these tests ourselves on off-the-shelf hardware, aiming to provide some take-away rules-of-thumb for choosing between a commercial and an open-source database system and optimizing the configurations for DSS queries. These rules can be used by any entreprise of this scale that will not run its own tests.

In the next sections, first we are going to examine which are the parameters that affect DSS performance as identified in literature, then we will explain the tests design and the rationale behind it, and finally we will present the tests results and discuss them.

4.1 Parameters Affecting the Performance of DSS Queries

We are interested in the characteristics of ad-hoc DSS workloads and the tuning parameters that affect them, for a given database size. DSS queries most of the times are special requests for managerial use, such as calculating the top salesperson last month or what products had the largest gains in sales last quarter. They tend to be highly complex and include a small number of large queries that involve large data scans, sorts and joins. On the other hand, they include very few, if any, updates.

Since DSS queries deal with large amounts of data within scans, sorts and joins, the size of the buffer pool and the sort buffer play an important role. Following the same logic, the fill factor and the page size can also contribute to having more data in the data cache and should, therefore, influence performance. Another option that would prove beneficial for large scans of data is intra-partition parallelism.

Morevoer, since there are very few updates, we can save some memory that would be allocated to the log buffer and schedule less frequent checkpoints. Finally, because DSS queries usually include only a small number of queries, we can reduce the number of active database agents as well as turn off intra-query parallelism. For the same reason, locking management and deadlock detection do not need to be very strict.

Modern database management systems have many configuration options that affect they performance (Shasha and Bonnet, 2002), however, in our experiments we will be intereted in the main parameters: buffer pool size, sort buffer size, fill factor, page size and intra-partition parallelism.

4.2 Implementation Decisions

The TPC-H specification document provides the basic rules that need to be applied in order to run a legitimate test with comparable results. However, it does allow some level of freedom in some aspects of the implementation. Below, we discuss the implementation decisions we made for our tests.

4.2.1 Refresh Functions

The refresh functions are not strictly defined by TPC, as their role is simply to ensure that the system under test is still able to execute basic updates in parallel with DSS query execution. As a consequence, we have some freedom in implementing them, following the basic pseudo-code provided. We chose to implement them as stored procedures.

4.2.2 Indexing

As explained before, the TPC-H benchmark (as opposed to TPC-R) involves an ad-hoc workload; it is aimed at unpredictable query needs. The rationale is: if you don't know what the query is going to be, you can't build a summary table or an index for it. TPC-H, therefore, allows indexes to be defined only for primary keys, foreign keys or columns of the date datatype. Specifically, TPC allows the definition of the primary and foreign keys of all tables, as well as a number of specified constraint checks. In addition, the standard allows single- or multiple-column indexes, provided that they reference the same table.

4.2.3 Partitioning

TPC-H also allows horizontal partitioning as long as the partitioning key is a primary or a foreign key or a datatype column. Partitioning used to split up large tables like Lineitem into multiple units enabling to parallelize I/O on aggregate (`group by`) queries. Unfortunately, our I/O setup does not feature multiple physical units.

4.2.4 Hardware

Our goal was to run the experiments on inexpensive off-the-shelf hardware, in order to simulate the most common conditions in smaller enterprises. We used an AMD Athlon processor with 1GB of RAM and a SATA 80 GB hard disk.

4.2.5 Scale Factor

Once again, in the interest of simulating the environment of a smaller enterprise, we chose the lowest possible scale factor yielding a 1 GB database. We believe that it is interesting to provide some results with a lower scale factor, as the only available ones to date are the official TPC-H results starting at 100 GB.

4.3 Parameters Varied in the Experiments

Our ambition was to test the two systems (SQL Server and MySQL) while varying the parameters previously identified as related to the performance of DSS queries. The first limitation was that we were running the tests on a single-core machine, in an attempt to stay true to our premise of using realistic off-the-shelf hardware; therefore, we could not evaluate the effect of intra-partition parallelism.

Furthermore, not all of the parameters can be set by the user in each of the database systems at hand. In SQL Server, it is not possible to set the size of the buffer pool or the sort buffer; only the total size of memory that the system can use can be set, by determining its minimum and maximum values. MySQL, on the other hand, allows to set up a specific size for the buffer pool and the sort buffer. Also, while SQL Server operates with a fixed page size of 8 KB, in MySQL the user can set the page size to 8, 16, 32 or 64 KB. Finally, in SQL Server it is possible to specify the fill factor for each page, while MySQL manages the free space automatically, with tables populated in sequential order having a fill factor of $^{15}/_{16}$. Table 3 display a summary with the names and values range of these parameters in MySQL 5.1 (SQL Server parameter list left out due to lack of space).

Table 3: Major tuning parameters affecting DSS query performance in MySQL 5.1.

MySQL 5.1				
	Min	**Max**	**Default**	**Parameter Name**
Buffer Pool Size	1 MB	4 GB	8 MB	innodb_buffer_pool_size
Sort Buffer Size	32 KB	4 GB	2 MB	sort_buffer_size
Fill Factor	n/a	n/a	(1/2 to) 15/16	n/a
Page Size	8 KB	64 KB	16 KB	univ_page_size

In light of these inconsistencies, we decided to run two general types of tests: the *memory size test* and the *number of rows per page test*. The tests are summarized in Table 4.

In the memory size test, we will be varying the total memory size in the two systems between 16MB and 1024MB; for MySQL, we will allocate each time $\frac{3}{4}$ of the memory to the buffer pool and $\frac{1}{4}$ to the sort buffer, as recommended by MySQL distributors (Oracle, 2005). Meanwhile, we will keep the fill factor and page size constant and approximately fair between the

Table 4: Parameters varied in the experiments for testing the influence of memory size and number of rows per page.

		SQL Server	MySQL
Mem Size Test	**Total Memory**	16-1024 MB	16-1024 MB (Buffer:sort 3:1)
	Fill Factor	90%	15/16 (default)
	Page Size	8 KB (default)	8 KB
Rows per Page Test	**Total Memory**	128 MB	128 MB (Buffer:sort 3:1)
	Fill Factor	40-100%	15/16 (default)
	Page Size	8 KB (default)	8-64 KB

two systems. For SQL Server, the page size is 8K by default and we will set the fill factor at 90%, which is almost equal to the MySQL default. Similarly, for MySQL, the page size is set at 8K and the fill factor is the default $^{15}/_{16}$ ($\approx 93\%$).

In the number of rows per page test, we will be varying the fill factor in the case of SQL Server and the page size in the case of MySQL. Note that the resulting range of number of rows per page is different for the two database systems, but that serves exactly the purpose of verifying whether allowing the user to specify much larger page sizes gives MySQL an advantage. Meanwhile, the total memory is set at a medium value of 128MB; here too, the allocation of memory in MySQL is 3:1.

In the next sections, we are going to present the results of running the full TPC-H test in each of the two systems, under the configurations presented in Table 4. The tests were executed multiple times and the time recorded is the average of the slowest runs. The scale factor we used was 1, yielding a 1GB database. Apart from performance in minutes, we will also show the values of the TPC-H metrics for each test. For these calculations, we considered the hardware cost to be approximately 500$ and the software cost the current price of 898$ for SQL Server 2008 and 0$ for MySQL 5.1.

4.4 Full TPC-H Tests in SQL Server

For the memory size test, we run seven tests varying the total server memory from 16 to 1024 MB, while keeping the fill factor at 90%. For the number of rows per page test, we run five tests varying the fill factor from 40 to 100 %, while keeping the total server memory at 128 MB. In both cases, the page size was the default 8 KB.

4.4.1 Varying the Memory Size

In the first tests with SQL Server , we keep the fill factor at 90%, which is a realistic value for DSS workloads with few updates, while varying the total server memory size. Both min server memory and max server memory parameters were set to the same value, to achieve a fixed memory size. As we change the value from 16 to 768 MB, the performance improves significantly. The system ends up reaching its full potential around 512 MB; moving to 768 MB does not make much difference; it looks like 512 MB of memory are just enough to allow the server to keep all useful pages in cache. Finally, when we run the test with 1024 MB of server memory, the performance suffered severely compared to the previous value of 768 MB. In this case the server tries to allocate all physical memory in the system. Part of the buffer cache will be on virtual memory causing pages to be swaped in and out incurring in further I/O operations. As a final note, in terms of the TPC-H price/performance metric, we can claim that increasing the memory size from 64MB to 768MB is equivalent to a 40.39% cost reduction. This is a cost reduction of 0.06% for every additional MB of memory, not taking into account the test outliers observed with memory size of 16 or 1024 MB. Table 5 holds the results for the memory size test in SQL Server.

Table 6: TPC-H full test results for increasing fill factor in MS SQL Server 2008. Results for 90% fill factor ommited to due to space constraints without any significant loss in detail.

MS SQL Server 2008 - Number of Rows per Page Test				
total memory	128 MB	128 MB	128 MB	128 MB
fill factor	40%	60%	80%	100%
page size	8 KB	8 KB	8 KB	8 KB
load test	27min	22min	20min	19min
perf. test	2h2min	1h9min	1h3min	59min
Power@1GB	24.86qph	117.41qph	119.42qph	120.01qph
Thrp.@1GB	48.13qph	54.64qph	66.83qph	69.89qph
QphH@1GB	34.59qph	80.10qph	89.34qph	91.58qph
PPQphH@1GB	40.42$	17.45$	15.65$	15.23$

4.4.2 Varying the Number of Rows per Page

Subsequently, we attempted varying the value of the fill factor parameter. At first we attempted varying the parameter from 40% up to 100%, while keeping the total server memory at 768 MB. The results were identical and seemed to indicate that the fill factor does not influence the performance. This was counter-intuitive, since the fill factor definitely plays a role in the total amount of data that can be held in the buffer pool: the buffer pool can only hold a specific number of data pages and each one of these pages carries as much data as the fill factor dictates. Therefore, we decided to run the tests again with the total server memory set at 128 MB. Indeed, this time varying the fill factor did have a major impact on performance. The explanation for this is that a high memory size

Table 5: TPC-H full test results for increasing memory size in MS SQL Server 2008.

MS SQL Server 2008 - Memory Size Test							
total server memory	16 MB	64 MB	128 MB	256 MB	512 MB	768 MB	1024 MB
fill factor	90%	90%	90%	90%	90%	90%	90%
page size	8 KB	8 KB	8 KB	8 KB	8 KB	8 KB	8 KB
load test	46min	20min	19min	17min	16min	16min	36min
perf. test	4h54min	1h13min	1h	52min	41min	40min	1h9min
Power@1GB	30.76qph	115.75qph	119.60qph	138.44qph	162.35qph	164.38qph	117.34qph
Throughput@1GB	11.90qph	53.30qph	68.04qph	75.60qph	105.32qph	105.67qph	63.08qph
QphH@1GB	19.13qph	78.55qph	90.20qph	102.30qph	130.76qph	131.80qph	86.03qph
Price-per-QphH@1GB	73.08$	17.80$	15.49$	13.67$	10.69$	10.61$	16.25$

allowed the server to keep all necessary pages in the buffer pool, even if a low fill factor meant more pages and more I/Os the first time they are fetched. However, a low memory size means that a limited number of pages can be kept in the cache and is then substituted by other pages, therefore the amount of data in each page is significant as it represents the total amount of data kept in the cache and can lead to fewer I/Os for page substitutions.

One more reason why the fill factor significantly influences performance is the fact that DSS queries tend to contain large table scans with high selectivity. That means that they tend to access the disk sequentially instead of randomly and, in most of the cases, fetching a data page to disk means taking advantage of a large portion of its data. Therefore, a higher fill factor ensures less I/Os. Moreover, a higher fill factor can cause slower inserts and updates; however, DSS workloads have very few updates and we are running the tests on a single-core machine, therefore these factors do not cause performance overhead.

In terms of the TPC-H price/performance metric, increasing the fill factor from 60% to 100% is equivalent to a 12.72% cost reduction. This is a cost reduction of 0.32% for every additional 1% of fill factor, not taking into account the test outlier of fill factor 40%. Table 6 holds the results for the number of rows per page test in SQL Server.

4.5 Full TPC-H Tests in MySQL

For the memory size test, we run seven tests varying the total server memory from 16 to 1024 MB, while keeping the page size at 8 KB. For the number of rows per page test, we run four tests varying the page size from 8 to 64 KB, while keeping the total server memory at 128 MB. In both cases, the fill factor was the default 15⁄16.

4.5.1 Varying the Memory Size

As with SQL Server, we observe that more space for caching data pages translates to performance improvements. Once again, the performance improves significantly when incrementing lower cache size and the effect becomes less visible as the size increases. Again, when the cache size reaches the available physical system memory, the performance suffers.

Like in SQL Server, tuning more appropriately leads to impressive reductions on the cost: 27.88% price/performance metric difference between the test with 64MB of total memory and that with 768MB. This is a cost reduction of 0.04% for every additional MB of memory, not taking into account the test outliers observed with memory size of 16 or 1024 MB. Table 7 holds the results for the memory size test in MySQL.

Table 8: TPC-H full test results for increasing page size in MySQL 5.1.

MySQL 5.1 - Number of Rows per Page Test				
buf pool size	96 MB	96 MB	96 MB	96 MB
sort buf size	32 MB	32 MB	32 MB	32 MB
fill factor	15/16	15/16	15/16	15/16
page size	8 KB	16 KB	32 KB	64 KB
load test	20min	18min	17min	17min
perf. test	1h13min	59min	52min	50min
Power@1GB	114.67qph	122.58qph	141.73qph	144.88qph
Thrp.@1GB	55.59qph	69.66qph	79.57qph	82.58qph
QphH@1GB	79.84qph	92.41qph	106.20qph	109.38qph
PPQphH@1GB	6.26$	5.41$	4.71$	4.57$

4.5.2 Varying the Number of Rows per Page

One could argue that a higher page size would not greatly influence performance, as in the best case scenario it means the same amount of data in the buffer pool only differently organized and in the worst case scenario it means more irrelevant data were fetched along with relevant ones. However, DSS queries tend to access data sequentially with large table scans; thus, it is highly likely that each page fetched contains many relevant rows. This means that a higher page size could lead to less I/Os during a scan. What is more, data pages include a page information preamble that occupies valuable space. Therefore, the less pages we divide the buffer pool size into, the less

Table 7: TPC-H full test results for increasing memory size in MySQL 5.1.

MySQL 5.1 - Memory Size Test							
buffer pool size	12 MB	48 MB	96 MB	192 MB	384 MB	576 MB	768 MB
sort buffer size	4 MB	16 MB	32 MB	64 MB	128 MB	192 MB	256 MB
fill factor	15/16	15/16	15/16	15/16	15/16	15/16	15/16
page size	8 KB	8 KB	8 KB	8 KB	8 KB	8 KB	8 KB
load test	48min	23min	20min	16min	16min	14min	57min
performance test	5h32min	1h28min	1h13min	1h2min	56min	54min	1h44min
Power@1GB	30.25qph	111.52qph	114.67qph	118.93qph	136.52qph	139.68qph	103.68qph
Throughput@1GB	10.02qph	51.39qph	55.59qph	67.76qph	78.73qph	79.08qph	48.12qph
QphH@1GB	17.41qph	75.70qph	79.84qph	89.77qph	103.67qph	105.10qph	70.63qph
Price-per-QphH@1GB	28.72$	6.60$	6.26$	5.57$	4.82$	4.76$	7.80$

space goes wasted in non-data preambles. Finally, depending on the record size, it is possible that a larger page size ensures that more data is stored per page: for instance, supposing fill factor 100%, if the record size is 4,5 KB, we can fit only one record in a 8 KB page but three records in a 16 KB page. This last reason could explain the big performance difference we observed between the test with page size 8 KB and the one with 16 KB. Continuing to increase the page size, we found further improvement in the total run time; however, less and less impressive. In terms of the TPC-H price/performance metric, increasing the page size from 8 to 64 KB is equivalent to a 27.00% cost reduction. This is a cost reduction of 0.48% for every additional KB of page size. Table 8 holds the results for the number of rows per page test in MySQL.

4.6 Discussion

Examining the first tests in the results tables, we can see that, for the same page size, approximately the same total memory size, and a fill factor of 90% and automatically controlled respectively, the performance of MySQL is slightly worse, altough of the same magnitude. This could mean that the automatic handling of the fill factor in MySQL cannot compete with the user-defined high fill factor in SQL Server, or that there are some more tuning parameters (besides the major ones which we strived to set fairly) that cause performance deterioration when left in their default values. Most likely, however, this performance difference indicates the superiority of SQL Server's query optimizer when dealing with highly complex DSS queries, which can also be seen on Figure 4. This conclusion is strengthened by the observation that for our setup both systems seem to reach their full potential after 512 MB of total memory as their performance stops depending on the amount of data in the buffer pool and becomes stable, and at that point SQL Server is ahead.

For the same total memory size, increasing the page size is more effective for DSS queries than increasing the fill factor. This makes total sense because, as we discussed, since we have full scans, relevant data tend to be next to each other; therefore, the more data per page the better the performance. A high fill factor in SQL Server can only achieve full use of the 8 KB data page, while a 32 or 64 KB page in MySQL will store a lot more data regardless the fill factor automatically assigned by the system. Thus the performance difference. As users, we can conclude that having control over the page size is a better tool for DSS queries performance enhancement than control over the fill factor. Hence, here the MySQL approach is superior. However, increasing the memory size has an influence that exceeds both those of increasing the page size or the fill factor.

One final conclusion that can be drawn by these results is that, even though the TPC-H tests run faster in SQL Server, the price/performance metric favors MySQL by far (see Figures 3b and 4a). The additional 898$[1] required to purchase an SQL Server 2008 license do not seem to be worthy for such low-scale database needs in terms of performance difference. Therefore, there is a trade-off between high performance with SQL Server and cheaper implementation with MySQL. Since the performance difference is not that huge, it makes sense that MySQL is chosen by small businesses. However, in a setting of a large corporation where multuple DSS queries are run concurrently, this performance difference will presumably justify the added price of the commercial SQL Server license.

5 CONCLUSIONS AND FUTURE WORK

After examining the TPC-H benchmark in detail and having ran the tests, we can conclude that the test can be downscaled. Even when using a low scale factor as we did in order to run the tests on cheap

[1] circa 2010.

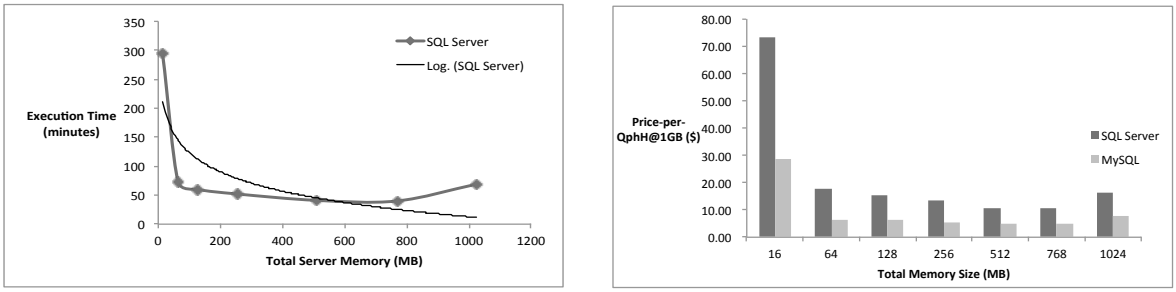

Figure 3: Impact of increasing the memory size parameter. Evolution of total execution on the left *(a)* and the evolution of the price/performance metric on the right *(b)* . The left graphic also displays the corresponding log fitting curve.

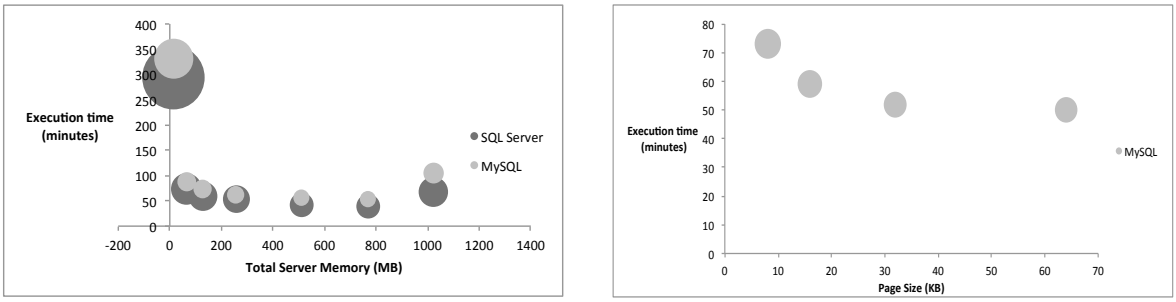

Figure 4: Relative influence of different parameters on execution time and cost. Influence of memory size on the left *(a)* and influence of page size for MySQL only on the right *(b)*. Larger bubles represent greater cost.

off-the-shelf hardware (thus yielding results that cannot be directly compared to the official TPC results) we could still observe differences between different systems and configurations. However our intuition is that the set-up time and complexity for the database schema, workload and tests make the benchmark an unlikely choice for a medium-sized enterprise without a team of experts that can produce the required code.

Running TPC-H gave us the motivation to look deeper into the factors that influence the performance of DSS queries. We experimented with SQL Server 2008 and MySQL 5.1 while varying such tuning options and parameters, arriving to some interesting conclusions. From Sections 4.4 and 4.5 we conclude that the most influential tuning option is undoubtely the memory size, namely the buffer pool and the sort buffer sizes, which results in great improvement in the price/performance metric. We observed clearly that other parameters such as the page size and the fill factor can also play an important role in the performance of DSS queries and hence influence the price/performance metric.

Since the two systems do not have identical configuration options, it is difficult to ascertain whether we tuned them fairly. Nevertheless, for similar configurations, MySQL is consistently slower than SQL Server (see Figure 3b and 4a). Since tuning is most probably not responsible for this difference, we attribute it to different query optimizer philosophies. In any case, MySQL might take a little longer to execute the TPC-H tests, yet it has a higher price/performance ratio thanks to being a freeware. In other words, if you are not going for optimal performance, it is certainly a viable and cheap alternative.

In the future, it would certainly be interesting to run the TPC-H test with a higher scale factor. Such results could be directly compared to the official ones and we could establish whether it is possible to recreate such performances using cheap off-the-shelf hardware. In addition, with a larger database we would be able to observe finer differences between different configurations and reach more conclusions.

Moreover, earlier we mentioned that DSS queries are influenced by concurrency options, such as intra-partition parallelism. Running the tests on a multi-core machine would permit us to explore the performance differences caused by this option. There could also arise a trade-off between increasing the fill factor or the page size in order to keep more data into the cache, and keeping them low so that fewer data is unavailable when a page is locked.

Finally, having explored the process in detail, we are interested in developing our own plug-and-play kit of source code for running the TPC-H benchmark on any off-the-shelf hardware.

REFERENCES

Chaudhuri, S. and Narasayya, V. (2007). Self-tuning database systems: a decade of progress. In *VLDB'07, 33rd Int'l Conference on Very Large Databases*. VLDB Endowment Press.

Darmont, J., Bentayeb, F., and Boussaid, O. The design of dweb. *Int'l Journal of Business Intelligence and Data Mining*.

DeSota, D. (2001). Characterization of i/o for tpc-c and tpc-h workloads. In *CAECW'01, 4th Workshop on Computer Architecture Evaluation using Commercial Workloads*. IBM Press.

Elnaffar, S. (2002). A methodology for auto-recognizing dbms workloads. In *CASCON'02, 2002 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press.

Elnaffar, S., Martin, P., and Horman, R. (2002). Automatically classifying database workloads. In *CIKM'02, 11th Int'l Conference on Information and Knowledge Management*. ACM Press.

Elnaffar, S., Martin, P., Schiefer, B., and Lightstone, S. Is it dss or oltp: automatically identifying dbms workloads. *Intelligent Information Systems*.

Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. (1994). Quickly generating billion-record synthetic databases. In *SIGMOD'94, 1994 ACM SIGMOD Int'l conference on Management of Data*. ACM Press.

Green, C.-D. (2002). *Oracle9i Database Performance Tuning Guide and Reference*. Oracle, a96533-02 edition.

Guehis, S., Goasdoue-Thion, V., and Rigaux, P. (2009). Speeding-up data-driven applications with program summaries. In *IDEAS'09, 2009 Int'l Database Engineering and Applications Symposium*. ACM Press.

Harizopoulos, S., Shkapenyuk, V., and Ailamaki, A. (2005). Qpipe: a simultaneously pipelined relational query engine. In *SIGMOD'05, 2005 ACM SIGMOD Int'l Conference on Management of Data*. ACM Press.

IBM (2004). *DB2 Universal Database V7 Administration Guide: Performance*. IBM, sc09-4821 edition.

Kandaswamy, M. and Knighten, R. (2000). I/o phase characterization of tpc-h query operations. In *IPDS'00, 4th Int'l Computer Performance and Dependability Symposium*. IEEE Computer Society Press.

Kitsuregawa, M., Goda, K., and Hoshino, T. (2008). Storage fusion. In *ICUIMC'08, 2nd Int'l Conference on Ubiquitous Information Management and Communication*. ACM Press.

Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S., and Wenisch, T. (2009). Disaggregated memory for expansion and sharing in blade servers. In *ISCA'09, 36th Annual Int'l Symposium on Computer Architecture*. ACM Press.

O'Neil, P., O'Neil, E., and Chen, X. (2009). The star schema benchmark (ssb). [Online; accessed Feb 2012].

Oracle (2005). *MySQL Reference Manual*. Oracle, 28992 edition.

Oracle (2006). Conducting a data warehouse benchmark. [White paper; accessed Feb 2012].

Packer, A. and Press, S. M. (2001). *Configuring and Tuning Databases on the Solaris Platform*. Prentice Hall PTR, Upper Saddle River, 1st edition.

Paulsell, K. (1999). *Sybase Adaptive Server Enterprise Performance and Tuning Guide*. Sybase Inc., 32614-01-1200-02 edition.

Poess, M., Smith, B., Kollar, L., and Larson, P. (2002). Tpc-ds, taking decision support benchmarking to the next level. In *SIGMOD'02, 2002 ACM SIGMOD Int'l Conference on Management of Data*. ACM Press.

Scalzo, B. (2007). *Top 10 Benchmarking Misconceptions*. Quest Software, 121007 edition.

Seng, J.-L. A study on industry and synthetic standard benchmarks in relational and object databases. *Industrial Management and Data Systems*.

Shasha, D. and Bonnet, P. (2002). *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, 1st edition.

Siqueira, T. L., Ciferri, R., Times, V., and Ciferri, C. D. A. (2010). Benchmarking spatial data warehouses. In *DAWAK10, 12th Iint'l Conference on Data Warehousing and Knowledge Discovery*. Springer-Verlag Press.

Software, Q. (2008). *Benchmark Factory for Databases*. Quest Software, 090908 edition.

Somogyi, S., Wenisch, T., Ailamaki, A., and Falsa, B. (2009). Spatio-temporal memory streaming. In *ISCA'09, 36th Annual Int'l Symposium on Computer Architecture*. ACM Press.

Vandierendonck, H. and Trancoso, P. (2006). Building and validating a reduced tpc-h benchmark. In *MASCOTS'06, 14th IEEE International Symposium on Modeling, Analysis, and Simulation*. IEEE Computer Society Press.

Wiese, D., Rabinovitch, G., Reichert, M., and Arenswald, S. (2008). Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning. In *CASCON'08, 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. ACM Press.