Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Μελέτη της επίδοσης των ad-hoc decision-support ερωτημάτων

Διπλωματικη Εργασια

της

ΑΝΝΑΣ Γ. ΘΑΝΟΠΟΥΛΟΥ

Επιβλέπων: Τιμολέων Σελλής
Καθηγητής Ε.Μ.Π.

Εργαστηριο Συστηματων Βασεων Γνωσεων και Δεδομενων
Αθήνα, Ιούλιος 2010

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Συστημάτων Βάσεων Γνώσεων και Δεδομένων

Μελέτη της επίδοσης των ad-hoc decision-support ερωτημάτων

Διπλωματικη Εργασια

της

ΑΝΝΑΣ Γ. ΘΑΝΟΠΟΥΛΟΥ

Επιβλέπων: Τιμολέων Σελλής
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 12η Ιουλίου 2010.

.....	.....	.....
Τιμολέων Σελλής	Ιωάννης Βασιλείου	Γεώργιος Στάμου
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2010

.....

Anna Θανοπουλου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Συστημάτων Βάσεων Γνώσεων και Δεδομένων

Ευχαριστίες

This thesis was conducted in the research unit DMIR of Inesc-ID, in Instituto Superior Tecnico in Lisbon, Portugal. I would like to take this space to gratefully thank my supervisors, professora Helena Galhardas and professor Paulo Carreira. Not only did they happily welcome me and guide me through all the steps of this journey, but also gave me the motivation to gain insight into the subject and end up loving it.
Θα ήθελα επίσης να ευχαριστήσω θερμά τον επιβλέποντά μου από πλευράς Ε.Μ.Π., καθηγητή κ. Τιμολέοντα Σελλή. Χωρίς την πολύτιμη βοήθειά του, δεν θα είχα μπορέσει να εκπονήσω αυτήν την εργασία στο εξωτερικό.
Τέλος, ευχαριστώ την οικογένειά μου για βοήθεια υλική, συναισθηματική και ανεκτίμητη.

Περίληψη

Τα decision-support ερωτήματα υποβάλλονται από ανώτερα στελέχη που θέλουν να ανακτήσουν στοιχεία από μία βάση δεδομένων ώστε να εντοπίσουν τις διαγραφόμενες τάσεις των οικονομικών αποτελεσμάτων της επιχείρησης και να βοηθηθούν στη διαδικασία λήψης αποφάσεων. Τα ερωτήματα αυτά είναι εξαιρετικά πολύπλοκα, διαχειρίζονται μεγάλο πλήθος δεδομένων και συνήθως υποβάλλονται απροειδοποίητα, δηλαδή ad-hoc. Συνεπώς, είναι αναπόφευκτο να απαιτούν πολύ χρόνο για να εκτελεστούν και να καθίσταται επιτακτική η βελτίωση της επίδοσής τους. Το πρώτο βήμα προς αυτήν την κατεύθυνση είναι η επιλογή ενός συστήματος διαχείρησης βάσεων δεδομένων. Στη συνέχεια, θα πρέπει να βελτιωθεί η επίδοση του συστήματος ρυθμίζοντας τις κατάλληλες παραμέτρους. Τέλος, θα πρέπει να διερευνηθεί αν υπάρχει κάποιος τρόπος να απλουστευτεί το έργο του βελτιστοποιητή ερωτημάτων, καθώς η βελτιστοποίηση ανάγεται σε εξαιρετικά σημαντικό ζήτημα όταν τα ερωτήματα είναι τόσο περίπλοκα. Ένα χρήσιμο μέσο που θα μπορούσε να χρησιμοποιηθεί για πειραματισμό πριν από τη λήψη όλων των παραπάνω αποφάσεων είναι το TPC-H benchmark για ad-hoc decision support ερωτήματα.
Αυτή η διπλωματική εργασία καθιστά μία προσπάθεια εμβάθυνσης στις αρχές του TPC-H benchmark και αξιοποίησής του για τη σύγκριση των συστημάτων διαχείρησης βάσεων δεδομένων SQL Server 2008 και MySQL 5.1. Ακόμη, θα προσπαθήσουμε να εξηγήσουμε τις παρατηρούμενες διαφορές στην επίδοση των δύο συστημάτων, παρατηρώντας τη συμπεριφορά τους υπό διαφορετικές ρυθμίσεις και αναλύωντας τη λογική που ακολουθεί ο βελτιστοποιητής ερωτημάτων κάθε συστήματος.

Λέξεις Κλειδιά

Ad-hoc, Decision-Support, TPC-H, Βελτίωση επίδοσης βάσεων δεδομένων, Βελτιστοποίηση ερωτημάτων

Abstract

Decision-support queries are submitted by higher management executives who need to retrieve data from a database server in order to draw a pattern of the company financial results and facilitate their decision making process. These queries are highly complex, handle large amounts of data and are usually submitted unpredictably, or else ad-hoc. Therefore, it is inevitable that they take a long time to execute and it becomes crucial to find ways to optimize their performance. The first step would be to choose the most efficient database management system; then, tune it appropriately; finally, identify any ways to facilitate the task of query optimization, as optimization becomes extremely important at such levels of query complexity. A useful tool that could be used to experiment before making all the above choices is the TPC-H benchmark for ad-hoc decision-support queries.

This diploma thesis constitutes an attempt to examine the TPC-H database benchmark in detail and use it to compare the Microsoft SQL Server 2008 and MySQL 5.1 database systems. Furthermore, we will try to explain the performance differences by observing the systems behaviour under different configurations, as well as by examining the logic behind query optimization decisions in each system.

Keywords

Ad-hoc, Decision-Support, TPC-H, Performance Tuning, Query Optimization

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Database Benchmarking

Given the wide offer of computer systems nowadays as well as their great complexity, it is crucial to determine which is the best choice for one's needs, in terms of both price and 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 computer systems performance.

In order to measure the performance of a computer system, it is necessary to define a *benchmark.* A benchmark is a standarized test that aims at comparing the performance of different systems under the same conditions. It has two majors components: the workload specification and the metrics specification. The *workload* is the assortment of tasks that the test comprises. The *metrics* are numeric quantities calculated using the values of certain parameters measured during the test. After defining the benchmark, we run the workload on different systems and compare the generated metrics.

Benchmarks need to be **domain-specific**, since different parameters constitute performance indicators in each domain. For instance, in a network system we are interested in the cost per transaction, while in a database system we usually measure the processing time and storage space [4]. What is more, benchmarks should meet some specific criteria. First of all, they should be **relevant**, that is to say they must include typical operations of the problem domain and measure the appropriate characteristics. Second, they should be **portable** in order to be easy to implement on different systems. Third, they should be **scaleable** as to apply both to small and large computer systems. Finally, they should be **simple** so that people can easily understand them [2].

In this thesis, the primary focus lies on a specific domain of benchmarking, *database benchmarking.* Database benchmarking intends to measure a database management system's performance under a carefully chosen workload and specific configurations. Database benchmarks are further categorised according to the predominant type of transaction

present in their workload. Some examples include the TPC benchmarks **TPC-A** (Online Transaction Processing including a LAN or WAN network) and **TPC-B** (Online Transaction Processing without network), the Wisconsin University's benchmark **Wisconsin** (Relational Queries), Jim Gray's **AS^{3}AP** (Mixed Workload of Transactions and Relational Queries), Patrick O'Neil's **Set Query Benchmark**(Complex and Reporting Queries), and R.G.Cattell's **Engineering Database Benchmark** (Engineering Workstation-Server).

There are several standard bodies for defining database benchmarks. The two most prominent are SPEC[1] and TPC[2]. *SPEC (Standard Performance Evaluation Corporation)* is a consortium of vendors defining benchmarks for the domains of science and workstations. *TPC (Transaction Processing Performance Council)* is a consortium of vendors defining benchmarks for transaction processing and database domains [2].

This thesis is going to include a thorough examination of a TPC benchmark, therefore it is meaningful to expand a little bit on this benchmark-defining standard body and its history. TPC has the goal of specifying objective benchmarks in order to support the customers in their decision making. It provides appropriate database-system "tests", aiming at making it easy for anyone to determine the price/performance ratio of available software and hardware. According to its mission statement, *the Transaction Performance Processing Council(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.* In other words, this council was formed in order to provide well-documented benchmarks that are not intended to favor any specific database system. The Council consists of companies and, as one would expect, mainly computer system ones. Decision making is ran by the Full Council. Each member-company has one vote and a two-thirds vote is required to pass any motion.

The first database benchmark was TP1, developed at IBM. However, the credibility of this test was doubted, as it was believed to favor IBM systems. The first attempt towards the definition of an objective benchmark was made by Jim Gray in 1985 [5]. His article outlined a test for Online Transaction Processing which was given the name DebitCredit. Unlike TP1, it specified a true system-level benchmark where the network and user interaction components of the workload were included. Nevertheless, praised as it may have been for its theoretical value, it was not unanimously adopted as the main database benchmarking guideline [2].

It was at this moment, in 1988, that the TPC was founded to provide a generally approved benchmarking standard. The very first benchmarks released were TPC-A[3], following the philosophy of DebitCredit, and TPC-B[4], after TP1. In fact, these benchmarks constituted an attempt to define the TP1 and DebitCredit tests in a more strict way, in order to en-

[1] http://www.spec.org/
[2] http://www.tpc.org/
[3] http://www.tpc.org/tpca
[4] http://www.tpc.org/tpcb

sure their general recognition. After these benchmarks, the Council recognised the need to publish specific benchmarks for Decision Support systems and On-Line Transaction Processing ones. The first benchmark for the former category was TPC-D[5] and for the latter TPC-C[6]. There have been some attempts to replace TPC-C with new benchmarks (such as TPC-W), but they were not as successful as hoped because this benchmark has proved to be realistic enough to serve its purpose. On the other hand, TPC-D was successed by TPC-H[7] and TPC-R[8], both Decision Support benchmarks. Currently, the Council has been developing a new benchmark called TPC-DS[9], which will combine the best features of both TPC-H and TPC-R and will introduce an even more complex workload in order to simulate the Decision Support computations more realistically.

1.2 Problem

We would like to determine whether it is possible to achieve a better price/performance ratio with an open source database management system than with a commercial one, using the same off-the-shelf hardware.

We are particularly interested in comparing MySQL with Microsoft SQLServer. There seems to be a lack of bibliography when it comes to comparing these database management systems using the popular decision support benchmark of TPC-H. Indeed, the TPC-H official results[10] include only one test on MySQL that is executed using special hardware and software provided by Kickfire[11]. There are no results available when testing the two database management systems under the same conditions.

Furthermore, we are interested in the TPC-H benchmark itself and would like to study its implementation and find out whether it is really a test that can be easily repeated at home, in an inexpensive machine and without a team of specialists.

1.3 Solution

We are going to execute the TPC-H test with MySQL 5.1 and SQL Server 2008, on the same hardware. We will also try to determine which tuning parameters influence the results for each one of the systems and run the tests while changing their values. In order to interpret the results, we will divide the TPC-H queries into categories and observe the performance differences for each database management system.

[5] http://www.tpc.org/tpcd
[6] http://www.tpc.org/tpcc
[7] http://www.tpc.org/tpch
[8] http://www.tpc.org/tpcr
[9] http://www.tpc.org/tpcds
[10] http://www.tpc.org/tpch/results
[11] http://www.kickfire.com/

1.4 Contributions

This work will provide us with the following contributions:

1. a description of all the steps necessary to execute the TPC-H test on an open source and on a commercial database management system
2. an evaluation of the level of expertise required to execute the TPC-H test
3. a full understanding of the TPC-H benchmark
4. study and justification of the effect of certain tuning parameters on the performance of the two database management systems when executing the TPC-H test
5. comparison of the systems in terms of performance and price/performance ratio
6. insight into the differences of the two query optimizers

1.5 Organization

We begin our discussion with chapter 2, where we will present a short overview of the scientific uses of the TPC-H benchmark. In chapter 3, we are going to present the main features of TPC-H. Chapter 4 is about system preparation for the test, including implementation decisions. Chapter 5 focuses on the exact procedure of running the tests and obtaining the results. Chapter 6 examines some basic concepts of performance tuning, and identifies the most influential tuning parameters for decision-support queries. Chapter 7 contains a detailed presentation of query optimization techniques, as well as the philosophies of SQL Server and MySQL query optimizers. Chapter 8 exhibits the results and interpretates them. Finally, Chapter 9 provides a conclusion with the thesis findings.

Throughout this document, we make reference to the TPC-H Standard Specification [1]. For readability reasons, we are not going to mention it every time.

Chapter 2

Related work

2.1 TPC-H as a Scientific Tool

The TPC-H benchmark is primarily used to compare different hardware, different database vendor software and different database software releases. One can view such official results of running the full TPC-H benchmark at the TPC website[1]. Further than that, there are some semi-official independent results, such as white papers [17], diploma theses [18] and small conference presentations [19].

Then again, the TPC-H method can be used, even partially, for the validation of a number of scientific ventures. There are many papers making such use of TPC-H. We are going to examine some of them, trying to cover as broad a scientific research area as possible.

We begin with two papers that make use of TPC-H only as a randomly populated database schema and execute their own workload over it.

First, we have a method of inferring table join plans in the absence of any metadata, such as attribute domains, attribute names or contraints [30]. The method enumerates the possible join plans in order of likelihood, based on the compatibility of a pair of columns and their suitability as join attributes, that is to say their appropriateness as keys. Two variants of the approach are outlined; one highly accurate but potentially time-consuming (exact method) and one less accurate but considerably more efficient (approximate method). In order to evaluate the two variants, the research uses the TPC-H schema randomly populated using the DBGEN utility and a query designed by the writers. The query executes a self-join of the largest TPC-H table, the table Lineitem, considering that to be the worst-case two-table-join query for the TPC-H schema. Various horizontal and vertical subsets of the Lineitem relation were considered, in an attempt to represent different relation sizes and numbers of candidate join attributes. During these tests, the exact method was tested for performance (that is to say, execution time) and the approximate for accuracy

[1] http://www.tpc.org/tpch

(or, correctness of the results set). The results of the experiments led the researchers to the conclusion that the approximate approach is several orders of magnitude faster than the exact one, while not being considerably less accurate.

In continuation, let us address the problem of minimizing the cost of exchanges between database programs and the database system [35]. In order to achieve this, this paper proposes the use of program summaries, which are graphs describing the sequence of accesses to the database that will be needed by the program. This way, the optimizer can manipulate data retrieval as a whole, instead of processing them as a flow of SQL queries submitted independently to the database system. Thereby, it becomes possible to choose a global strategy that minimizes the cost of transferring data from the server to the client. For the evaluation of the approach, the writers used the randomly populated database schema and executed three multi-query programs of their own design against it. In order to compare the performance difference while using and while not using the program graphs, the writers measured the total execution time of the programs.

Now let us examine some works in which the TPC-H schema as well as a selection of TPC-H queries are used to verify the superiority of the proposed technique. The criteria for choosing only a handful of TPC-H queries vary from paper to paper. A popular practice seems to be selecting queries that exhibit prevailing scan, join or balanced behavior (more details on this categorization will be given in chapter 2). Let us examine two papers that make use of this categorization and, in addition, are primarily interested in measuring the query execution time.

According to the first of these works, in contrast with common intuition, aggressive work sharing among concurrent queries does not always improve performance in a multi-processor environment [34]. In fact, as the number of cores in the system increases, a trade-off appears between exploiting work-sharing opportunities and the available parallelism. In order to prove the existence of this trade-off, the writers experiment with a selection of scan- and join-bound TPC-H queries against the TPC-H database schema, measuring the execution time observed when sharing and when not sharing work amongst queries. The paper concludes that sometimes it is more efficient to perform the same task in parallel than to perform it once and then apply it to each client's needs.

The next research is an attempt to re-examine traditional compute-memory co-location on a system and it details the design of a new architectural building block, the memory blade [37]. Memory blades allow memory to be disaggregated across a system and can be used for memory capacity expansion and for sharing memory across servers. Experiments with various benchmarks on implementations of this principle demonstrate that memory disaggregation provides substantial performance benefits. Among the benchmark experiments conducted is that of running three TPC-H queries, representing scan- and join-bound as

well as balanced behaviors, against the randomly populated TPC-H database schema. The experiments aim to measure the total execution time of the queries with or without the use of memory blades in the system design.

Continuing the presentation of works that select only a few TPC-H queries for their experiments based on the scan/join/balanced categorization, we will summarize two more papers, only this time the writers are interested in measuring values other than the execution time.

The first paper highlights the importance of in-page placement for high cache performance [32]. It shows how traditional techniques fail to utilize cache in an efficient way and proposes a new data organization model called PAX (Partition Attributes Across). This model significantly improves cache performance by grouping together all values for each attribute within each page. Because PAX only affects layout inside the pages, it does not slow down I/O or require more storage space. The approach uses a number of workloads to experimentally prove its superiority. Among those, two join- and two scan-bound TPC-H queries are used to corroborate the hypothesis that PAX ensures better cache performance. To make their point, the writers measure the cache hit ratio during the queries execution.

Carnegie-Mellon researchers have proposed two novel memory system designs to improve memory hit numbers, called temporal memory streaming and spatial memory streaming. The first design exploits the fact that memory addresses are temporally-correlated, which means that previous sequences of misses are likely to repeat. It replays previously observed miss sequences to eliminate long chains of dependent misses. The second design exploits the fact that memory addresses are spatially-correlated, that is to say local data tend to be relevant. It predicts repetitive data layout patterns within fixed-size memory regions. It is obvious that each one of these techniques targets a different subset of misses. Spatio-temporal memory streaming [33] exploits the synergy between spatial and temporal streaming. For the evaluation of this technique, many tests are run, including which the execution of three TPC-H queries that demonstrate intense join or scan behavior, or a balance between the two. Conclusions are drawn from the number of read misses that occured during the queries execution.

Now we proceed in examining works that choose to use a number of TPC-H queries following criteria different than the scan/join/balanced categorization.

There has been an effort to implement a simultaneously pipelined relational query engine, called QPipe [28]. The motivation for this new engine was based on the observation that concurrent queries often exhibit high data and computation overlap, such as accessing the same relations on disk, computing similar aggregates or sharing intermediate

results. However, query engines treat queries independently, only making sure that the available resources are efficiently allocated. This work proposes a query engine that proactively coordinates same-operator execution among concurrent users, thereby exploiting common accesses to memory and disk as well as common intermediate result computation, while not incurring additional overhead. The paper experiments with the randomly populated TPC-H database schema, running the eight TPC-H queries that access exclusively the three largest tables, Lineitem, Orders and Part. The queries are organized into randomly-sequenced workloads that run against QPipe and an unspecified major commercial database management system. The goal of these experiments is to demonstrate the difference in total throughput time when applying pipelining techniques to a number of operators that are dominant in each of these workloads. The justification for using TPC-H queries to this end is that they spend much time on scans and joins, and they can be generated using different predicates to provide suitable experimental randomness. The results reveal a clear throughput difference favoring the QPipe engine and becoming more pronounced as more clients are added.

The next work is interested in improving performance of column-oriented database systems, that is to say systems in which each attribute is physically stored as a separate column allowing queries to load only the required attributes [29]. The challenge for such implementations is reducing the cost of tuple reconstruction, which corresponds to joining two or more columns and is prompted by multi-attribute queries. This paper proposes partial side-ways cracking, a novel design that minimizes the tuple-reconstruction cost in a self-organizing way. In order to do so, it uses auxiliary self-organizing data structures called cracker maps which provide a direct mapping between pairs of attributes used together in queries for tuple reconstruction. Based on the workload, these maps are continuously kept aligned by being physically reorganized during query compilation phase, allowing the database system to handle tuple reconstruction using cache-friendly access patterns. Partial side-ways cracking is implemented on top of an open-source column-oriented database system dubbed MonetDB, and compared to row-oriented MySQL with analogous pre-sorting, using the TPC-H database schema randomly populated and a workload comprised of the twelve TPC-H queries that have at least one selection of a non-string attribute (because string cracking is not addressed in this paper). As the workload is run repeatedly using different values for query predicates each time, the execution time for each query is significantly reduced for MonetDB but remains constant for MySQL.

Finally, here are some works in which the criteria for choosing only some of the TPC-H queries are not specified.

Autonomic Tuning Expert is a framework for autonomic database tuning [31]. Autonomic tuning is aimed in reducing the cost of database administration as well as the possibility of human error while tuning a system. The paper is first interested in automatically

identifying different types of workloads and problematic scenarios. Then, it formalizes database administration knowledge about tuning for such cases, proposing tuning plans for each of them. The writers end up developing a reference system for autonomic tuning for IBM DB2 and consequently evaluate its performance using TPC-H queries as representative decision-support ones and TPC-C queries as representative online transaction ones. In fact, the workload consists of TPC-C queries running for a fixed amount of time, interrupted by TPC-H queries that run for the same amount of time and more TPC-C queries for the same amount of time. Thus, the system performance is not evaluated by the total execution time rather than by the measurement of values such as the buffer pool hit ratio and the number of row reads.

Storage fusion is the idea of deep collaboration between storage and database servers [36]. If we elegantly cut some portion of the software functions from the server and put it to the storage system, the storage system could work much more closely with the server. To achieve this, the paper proposes two techniques. First, the exploitation of query execution plans to enable dynamically informed prefetching. This implementation is evaluated by running TPC-H query number 8 and measuring the execution time with as well as without prefetching. The reasons for selecting query 8 are not specified. The second technique is putting autonomic database reorganization into the storage to relieve the management burdens of the database system. To highlight the importance of such implementation, the researchers gradually execute a large number of updates against the TPC-H database and observe the difference in execution time for 10 TPC-H queries after each set of updates. Thus, they prove the degradation of execution time due to data structural deterioration. The reasons behind the selection of these ten TPC-H queries are not revealed. Lastly, the research evaluates the total reorganization time for equally-sized TPC-H database instances in a system using the classic reorganization technique and a system using the new approach.

Chapter 3

The TPC-H Decision Support Benchmark

3.1 Benchmark Overview

The TPC-H benchmark was developed by the TPC and officially approved in 1999. A single-sentence description of this database benchmark would be that it is a *decision support benchmark comprising a suite of business oriented ad hoc queries and a few data modifications*. The keywords here would be *decision support* and *ad hoc* . This section aims at explaining these terms.

The TPC-H benchmark models the activity of an international product supply corporation. The business environment of such modeled business is divided in two large areas: the *business-operations area* and the *business-analysis area*. The business-operations area models the operational end of the business environment where transactions are executed on a real time basis. Benchmarks that measure the performance of systems managing this area are called *online transaction processing (OLTP)* benchmarks. Such benchmarks are TPC-C and TPC-W. On the other hand, the business-analysis area is where business trends are computed and refined data are produced to support the making of sound business decisions. Benchmarks focused on this area are called *decision support (DSS)* benchmarks. TPC-H and TPC-R are such benchmarks. Figure 3.1 illustrates the business environment.

The term *decision support* implies that higher management executives would need to retrieve data from the database in order to draw a pattern of the company financial results and facilitate their decision making process. Most commonly, the business analysis is centered around pricing and promotions, supply and demand management, profit and revenue management, customer satisfaction studies, market share studies and shipping management. In order to retrieve this kind of data from the database of a multinational corporation, one has to execute highly complex queries and deal with a large volume of data. For instance, one might wish to rank the company's customers according to various

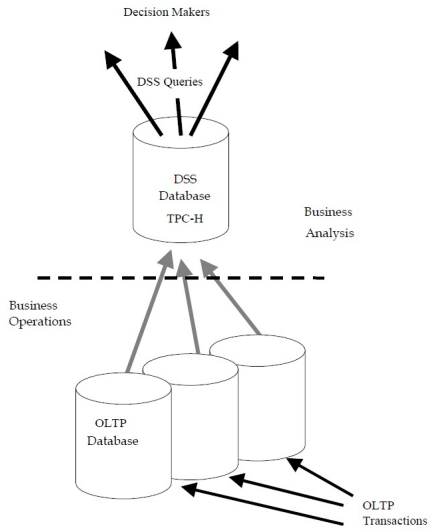


Figure 3.1: Business environment.

criteria such as the total ordered quantities or the order frequency, in order to reward the most important clients with a promotional offer. This would imply executing various predicates against a very large table containing the customers and possibly other tables containing information on orders. Also, the database has to be available for query execution on a 24/7 basis and it has to support multiple end-users as well as data modifications, since a real-world database is not a one-time snapshot of itself. The TPC-H benchmark constitutes an attempt to model such a database along with this kind of realistic business questions.

Both TPC-H and TPC-R are decision support benchmarks and use the same database schema and test the same queries. However, due to TPC-H's more realistic approach of the business DS environment, it has prevailed over TPC-R. This is due to the fact that the TPC-H benchmark involves an *ad-hoc workload*. That is to say, it is aimed at unpredictable query needs or else it does not presume prior knowledge of the queries to be executed. On the contrary, TPC-R judges predictable data retrieval and extraction, such as periodic reports. Therefore, TPC-R's workload is a *reporting workload*.

In the rest of the chapter, we are going to present the TPC-H schema and workload.

3.2 The TPC-H Schema

As mentioned before, the goal of TPC-H is to portray the activity of a wholesale supplier. However, instead of representing the activity of any particular business segment, the benchmark models any industry that manages, sells or distributes products worldwide, such as car rental, food distribution, parts or suppliers. Although the TPC-H specification only gives us the schema as a collection of tables, we tried to represent it as an E-R diagram in Figure 3.2, for best understanding of the model.

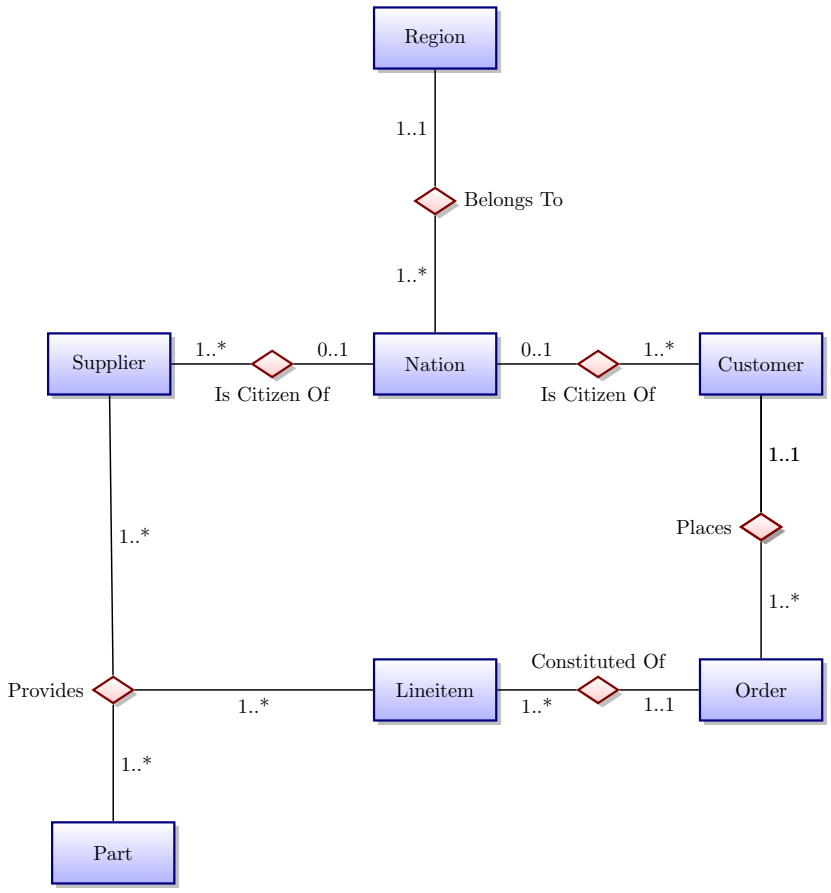


Figure 3.2: E-R diagram of the TPC-H Database.

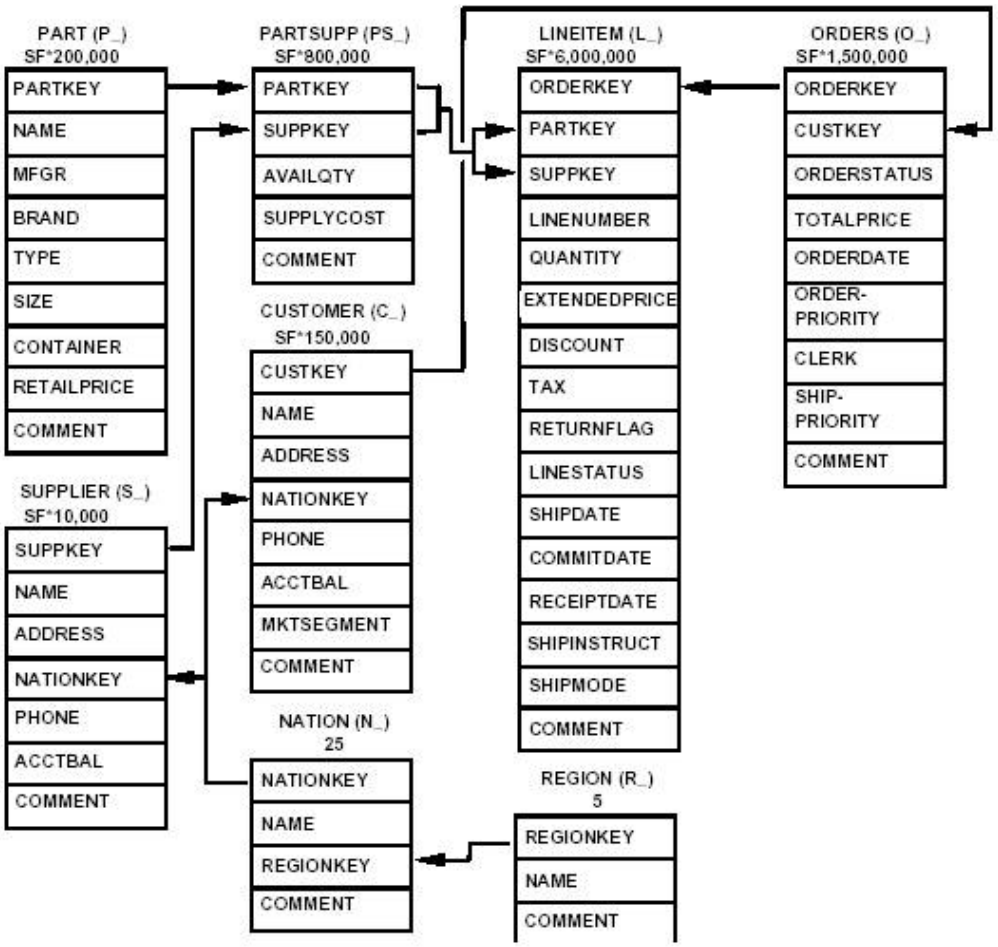


Figure 3.3: The TPC-H database schema.

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. The order has been saved as an invoice, that is to say a series of lines dedicated to one type of product each, which are represented by the **Lineitem** entity. As a result, each order is constituted of lines of items that 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**.

This E-R diagram is converted into a relational schema comprised by eight base tables, as specified by TPC. It is presented in Figure 3.3.

The two largest tables are Lineitem and Orders and contain about 83% of the total data.

3.3 The TPC-H Workload

The workload of the benchmark consists of 22 queries and 2 update procedures, all representing frequently asked decision making questions. 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 have a high-level of complexity and give answers to real-world business questions. Some of the classes of business analysis that they simulate are pricing and promotion, supply and demand management, profit and revenue management, customer satisfaction study, market share study and shipping management. 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 on the part of the database server. What is more, all queries are different from each other and have an ad-hoc nature. As we will explain in Chapter 4, each query runs stand-alone to demonstrate the ability of the test system to use all of the resources for a single user, as well as in multiple concurrent sessions to demonstrate the ability of the system to use all of the resources to satisfy concurrent users.

The TPC-H queries can be divided into 3 categories: *Scan-bound, Join-bound* and *Balanced behavior*[3]. This categorization is based on the total time each query spends on table scans and joins. An estimation of these times can be retrieved through the execution plan of the query. Scan-bound queries spend over 95% of their execution time on table scans. Exactly 50% of the TPC-H queries fall under this category. Such example is query number 1. For Join-bound queries, over 95% of the execution time is due to joins. About 25% of the TPC-H queries are Join-bound. Take query number 2, as an example: there are many tables to be joined, which explains the observed behavior. A query is considered to have Balanced behavior if 75% of its execution time is estimated to be due to table scans and 25% due to joins. The remaining TPC-H queries are categorized as queries with Balanced behavior. An example would be query number 17.

Chapter 4

Setting up the Test System

As mentioned in Chapter 1, the main objective of this thesis is to run the TPC-H benchmark test in order to produce comparable results for two database management systems and discuss the reasons behind the performance differences. There were many decisions to be made concerning the exact characteristics of the test system and the tuning of the database management systems. Actually setting up the test system for each one of the two database management systems includes: *(i)* creating the database with the exact schema proposed by the Transaction Performance Council; *(ii)* adding constraints; *(iii)* generating the flat data files using the DBGEN tool; *(iv)* loading the data into the tables using a script in order to measure the load time; *(v)* creating indexes and statistics; *(vi)* generating the workload queries to be executed using the QGEN tool; and *(vii)* installing the necessary stored procedures. Having completed these steps, we will then be able to run the workload and measure the execution times.

In this chapter, we begin by explaining the process of generating the test data with DBGEN and the queries with QGEN. Then, we proceed by listing the available options for implementing the database schema and configuring the system, as well as the decisions made respectively.

4.1 Database-Data Generation using DBGEN

DBGEN is a data generator provided in the TPC-H package to help fill all tables with a large amount of appropriate random data. The user selects the *scale factor* he prefers: the size of all tables, except for nation and region, scales proportionally with the scale factor. The available scale factors are:

1, 10, 30, 100, 300, 1000, 3000, 10000, 30000, 100000

The database size is defined with reference to the scale factor. For instance, choosing the scale factor to be equal to 1 means there will be generated 1GB of data in total.

For each column datatype, DBGEN follows a different *grammar*. A grammar in computer science consists of a set of rules for string structure. These rules specify which type of word (or elements) can be added in which part of a string so that the latter is valid under this grammar. DBGEN follows the grammar for each datatype, producing large amounts of random strings of the correct style. The elements come from a big bank of data containing words and numbers.

Other than table population, DBGEN is also used to produce random used by the refresh functions when they add lines to already populated tables. The method for this is the same as before.

4.2 Query Generation using QGEN

The 22 TPC-H queries are in fact 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 function to be performed by the query. However, this definition is not complete; there is a need to fill in some *substitution parameters* in order to complete the query syntax. The substitution values 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 the method used by DBGEN. After running QGEN, we get ready-to-run queries in valid SQL, which have the same basic structure as the query templates but vary in a random way when it comes to some predicate parameters.

In order to validate the results of a query, the TPC-H specification accompanies each query by its validation output data for a specific value of the substitution parameters upon a 1GB database. Obviously, this cannot be done for queries generated with random values in the substitution parameters' places.

The refresh functions are not strictly defined by TPC, as their role is simply to ensure that the system under test can execute basic updates in parallel with query execution. As a consequence, we have some freedom in implementing them, following the basic pseudo-code provided. They were implemented as stored procedures. The corresponding source code can be found in Appendix B.

4.3 Implementation Decisions

As explained in Chapter 2, the TPC-H benchmark (as opposed to TPC-R) involves an ad-hoc workload; it is aimed at unpredictable query needs. The test designers' reasoning is this: if you don't know what the query is going to be, you can't build

a summary table or an index for it. The TPC-H, therefore, allows indexing only on primary keys, foreign keys or date columns. This benchmark was designed to measure the database engine's ability to cope with queries that are not known in advance. Therefore, the TPC-H Specification sets strict rules about optimization.

4.3.1 Indexes

TPC allows indexes on one or more columns in no more than one table. These columns must be either a primary key, or a foreign key, or part of a compound primary/foreign key, or an attribute of "date" datatype. The reason behind these restrictions is the fact that the TPC-H method simulates an ad-hoc decision support workload, as explained in chapter 3. Therefore, wanting to preserve this spirit, we decided to create all indexes allowed. The complete set of index creation statements appears in appendices B and C under database build scripts.

4.3.2 Constraints

We defined all primary keys, foreign keys and check constraints allowed by the TPC-H specification document. Defining primary keys on all tables automatically set off the creation of the corresponding indexes. In the case of foreign keys however, we needed a second step in order to create a clustered index for them.

4.3.3 Horizontal Partitioning

According to the TPC-H Specification, horizontal partitioning is allowed as long as the partitioning key is a primary or a foreign key or a "date" column. However, since we are running the tests on a single-core machine, we cannot have intra-partition parallelism and thus we are not going to make use of this option.

Chapter 5

Running the Tests

Running 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. Naturally, the load test has to precede the performance test. In this chapter, we will describe these two tests and then list the metrics used to compare performance between systems.

5.1 The Load Test

After creating the database and generating the data files using the DBGEN tool, we can go ahead and execute the *load test*. This includes all the necessary steps between database creation and running the performance tests. 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 we settled on in the previous chapter, calculate the statistics for these indexes and install the refresh functions as stored procedures. The exact steps to be completed for the load test are illustrated in Figure 6.1.

As seen in Figure 6.1, the *database load time* is the necessary time to complete the steps of table creation, data loading, constraint addition, index creation, statistics calculation and stored procedures installation. All these steps are executed using a single script per database management system. The corresponding source codes are presented in Appendices B and C. The database load time is an important result to be reported, as it pictures the database management system's efficiency in setting up and populating a database.

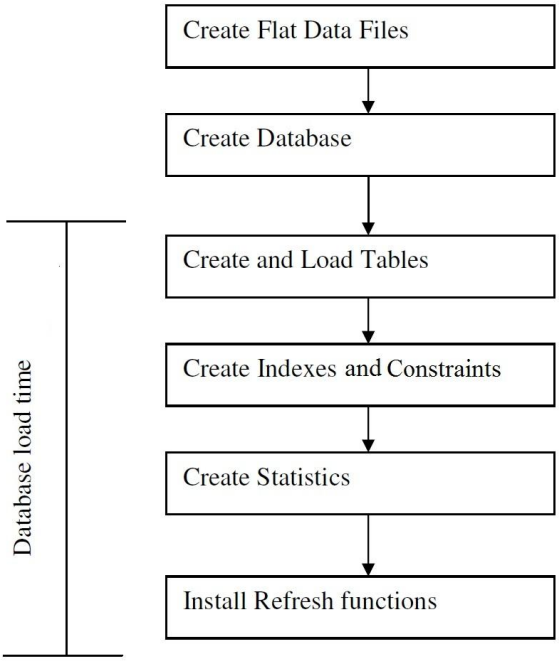


Figure 5.1: Steps for the Load Test.

5.2 The Performance Test

As soon as we complete the load test, we proceed with the performance test. It consists of two *runs*. Each run is an execution of the *power test* followed by an execution of the *throughput test*. Figure 5.2 illustrates the steps for running a complete sequence of the two TPC-H tests. The source codes of the scripts that execute the performance test for each database management system can be found in Appendices B and C.

In order to define the terms of power and throughput tests, we need to introduce the concept of *sessions*. A session is either a *query stream*, that is to say a sequential execution of each of the 22 TPC-H queries, or a *refresh stream*, which is a sequential execution of a number of pairs of refresh functions. Tests will consist of query streams and refresh streams.

5.2.1 Power Test

The purpose of the *power test* is to measure the raw query execution power of the system when connected with a single active user, that is to say how fast can the system compute the answer to a single query. This is achieved by running a single query stream session, 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.

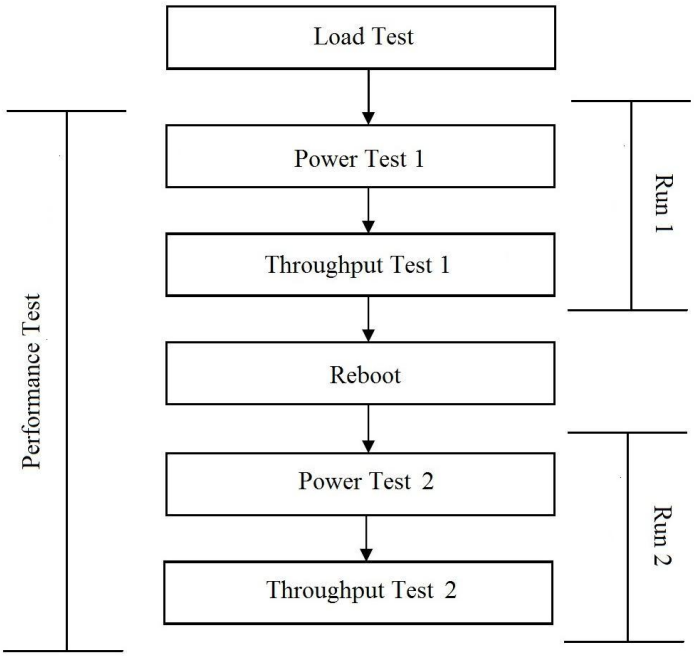


Figure 5.2: Steps for the TPC-H tests.

In particular, there are three steps necessary to implement the power test: *(i)* execution of the refresh function 1 by the refresh stream, *(ii)* execution of the query stream and *(iii)* execution of the refresh function 2 by the refresh stream. 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.

5.2.2 Throughput Test

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. In other words, this test is used to demonstrate the performance of the system against a multi-user workload; we want to measure exactly how scalable the system is. 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 5.1. What is more, the throughput test must be executed in parallel with a single refresh stream session. The number of refresh function pairs in this stream has to be equal to S. Each query stream and refresh function pair in the throughput test has an ordering number represented as **s** and ranging from 01 to S.

Like in the power test, the execution sequence for the queries in a query stream is pre-defined by TPC and determined by its 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. The TPC has come up with these sequences using a algorithm for random numbers generation.

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

Table 5.1: Number of query streams(S) (on the right) for a given scale factor(SF) (on the left).

5.3 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 the results in three forms: the database load time as discussed, the *measurement interval* and the *timing intervals*. The measurement interval represented as $\mathbf{T}_s$ is the total time needed to execute the throughput test. The timing interval represented as **QI(i,s)** for a given query Q_i is the execution time for the query Q_i within the query stream s, where s is 0 for the power test and the ordering number of the query stream for the throughput test. The timing interval **RI(j,s)** is the execution time for the refresh function RF_j within a refresh stream s, where s is 0 for the power test and the position of the pair of refresh functions in the stream for the throughput test. All these results must be measured in seconds, as specified by the TPC.

Next, we have to combine these results to produce some 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 a 1GB database. This metric equally weights 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. In the following, **SF** stands for scale factor.

5.3.1 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 timing intervals QI(i,0) and RI(j,0), that is to say the execution times for each one of the queries and the refresh functions obtained during the power test. We remind that the query and refresh streams in the power test have ordering number s that is equal to 0.

The geometric mean is a type of mean or average that indicates the central tendency of a set of numbers. It is similar to the most commonly used arithmetic mean, except that instead of adding the set of numbers and then dividing the sum by the count of numbers in the set, 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, $\frac{1}{2}$, $\frac{1}{4}$ is the cube root of their product ($\frac{1}{8}$), which is $\frac{1}{2}$; that is $\sqrt[3]{1 \times \frac{1}{2} \times \frac{1}{4}} = \frac{1}{2}$. The geometric mean can also be 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.

In this case, some of the query execution times are substantially different from the rest, meaning that they are significantly too long or too short. This fact would influence the arithmetic mean unduly, therefore is preferable to use the geometric mean. Let us assume, for instance, that we have only 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 Power@Size 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 is the geometric mean of the timing intervals for the 22 queries and the 2 refresh functions, a total of 24 factors. It represents the 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 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 × scale-factor.

5.3.2 Throughput Power Metric

The *throughput power metric* represented as **Throughput@Size** is computed as the ratio of the total number of queries executed within all the query streams of the throughput test, over the length of the measurement interval T_s. In simpler words, this metric tells us how many queries were executed in the elapsed time.

The Throughput@Size metric is defined as:

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

The numerator is the total number of executed queries within all streams (S streams with 22 queries each) and the denominator is the total time for the test. Therefore, the fraction represents the number of queries executed per second. Multiplied by 3600 seconds, it gives the number of queries executed per hour. Then we multiply this result by the scale factor in order to get the Throughput@Size, where size is the GB implied by the scale factor. The units are queries-per-hour × scale-factor, same as in the case of Power@Size.

5.3.3 The Composite Query-Per-Hour Performance Metric

The *composite query-per-hour performance metric* represented as **QphH@Size** combines the values of the corresponding metrics Power@Size and Throughput@Size.

The QphH@Size metric is defined as:

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

This metric is obtained from the geometric mean of the previous two metrics. By combining the values of the corresponding metrics, the metric expresses the overall performance level of the system, both for single-user mode and multi-user mode.

As a last note, since the TPC-H metrics reported for a given system must represent a conservative evaluation of the system's level of performance, the reported performance metrics must be for the run with the lower composite query-per-hour metric.

5.3.4 The Price/Performance Metric

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 met-

ric.

The Price-per-QphH@Size metric is defined as:

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

The symbol $ stands for the total system price in the reported currency. The units are the currency units, such as $.

This last metric will allow us to make the final price/performance comparison between the commercial and the open-source database management systems and comment on whether we are getting "our money's worth" when using the commercial one.

Chapter 6

Performance Tuning for Decision Support Workloads

6.1 Performance Tuning basics

Database administration textbooks define *performance tuning* as the act of customizing the available settings and configuration in order to maximize the use of resources and ensure efficient as well as rapid performance [20] [41]. There are three areas of tuning: system tuning, database tuning and application tuning. System tuning is the highest level, which means that system problems causes all databases and applications to perform poorly. Following the same logic, a database problem causes related applications to perform poorly. This hierarchy is better illustrated in Figure 6.1.

6.1.1 System Tuning

System tuning refers to the overall system, comprising the database management system itself as well as any components and software on which it relies, such as memory, disk, CPU, the operating system and the networking software. Installation, configuration and connectivity issues must be resolved appropriately in order to achieve optimum performance. We are going to work with a given set of hardware and software components, but we can still configure the database management system in an optimal way. In order to do so, we are going to examine various parameters that manage memory usage, number of active database agents and locking configurations.

There are multiple *cache memories (or buffers)* utilized by the database system to reduce the cost of I/O by avoiding redundant I/O operations1. Efficiently allocating the available memory resources to them is extremely important. Cache memories include the data cache, the procedure cache, the sort cache and the database log

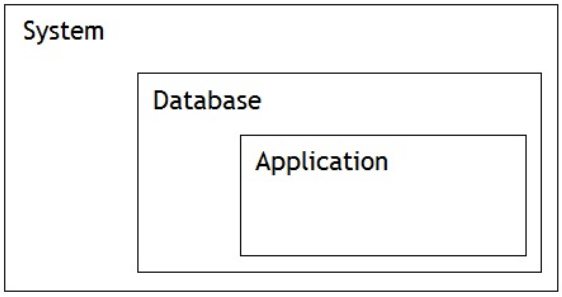


Figure 6.1: The tuning boxes: application performance is bounded by database performance which is in turn bounded by system performance.

cache.

The *data cache (or buffer pool)* stores table data pages as they are being read from disk. The next time an application requests data, the database system looks for it in the data cache before accessing the disk. If it finds it, we have a *hit*; otherwise, it's a *miss*. There are ways to measure the data cache *hit (or miss) ratio*. Since memory access is much faster than disk access, the less often the database system needs to read from or write to a disk, the better the performance. Thus, it is beneficial to have as much memory allocated to the buffer pool as to not oppose to the rest of the system's requirements.

The *procedure cache* stores queries and the corresponding query plans produced by the optimizer (see chapter 7 for details on optimization). Thus, the, frequently long, optimization process need not be performed every time a query is run, resulting in better performance. The administrator should try to allocate enough memory to this cache to store several recent query plans.

The *sort cache* stores intermediate sort results. Many operations such as grouping, ordering and union involve sorting. Therefore, the bigger the sort cache, the more sorting functionality can be performed in memory and the better the performance since I/Os are expensive.

The *database log cache* stores all changes made to the database. Usually, there are two log caches, one for log writes and one for log reads. The *log write cache* stores the changed data, which are over time written asynchronously to disk. Thus, we can speed up database modifications. The database system specifies a *system checkpoint interval* to guarantee that all log records are written safely to disk. The *log read cache* is used for rollback or recovery operations, when it is necessary to undo or reapply database changes. For data safety reasons, it is important to ensure that there is enough space in the log caches for a large number of recent updates.

Returning to system tuning, the *active database agents* are database clients cur-

rently connected with the database system. There is always a configuration option for setting the number of concurrent active database agents. Setting this option depends on the usual number of users of the database.

Finally, since database queries undergo processing that breaks them down to individual *operations* that require the use of some CPU or I/O component, it is important to handle correctly the exclusive *locking* of those resources for one transaction at a time. This practice ensures consistent data but it can also lead to considerable execution delays due to lock suspensions, timeouts and deadlocks. *lock suspensions*, which occur when a transaction requests a lock to a busy resource, *timeouts*, which occur when a transaction is terminated because it has been suspended for longer than a preset interval and *deadlocks*, which occur when two or more transactions cannot continue processing because each is waiting on a resource held by the other. In order to enhance performance, we have to try to avoid all these causes of delays and execution fails. For instance, the database system frequently checks for deadlocks. How often this happens, that is to say the length of the *deadlock detection cycle*, is subject to configuration. Setting this length to a small value (that is to say, frequent checks) guarantees better safety for the running applications; on the other hand, this check occupies valuable resources and it may interfere with the execution performance.

6.1.2 Database Tuning

Database tuning includes the physical design of the database as well as close monitoring for performance degradation due to file growth and disorganization. The most important aspect of database tuning is indexing, that is to say keeping an alternate path to data in the database sorted according to the value of one or more table columns. However, since TPC-H imposes specific rules on index creation as explained in chapter 4, we are going to focus on the rest of the influencing parameters. These include partitioning and reorganization, as well as management of free space and page size.

Partitioning consists of breaking a table into sections stored in multiple files. This can be done horizontally (based on the rows) or vertically (based on the columns). Whenever the files are stored in independent physical devices, partitioning helps to accomplish *parallelism*, meaning to allow the same request to split into multiple requests and utilize different CPU or I/O components in parallel. This option makes a performance difference in multi-core environments that handle data in large relations and perform frequent scans.

Free space (or fill factor) can be used to leave a portion of a table space empty and available to store newly added data. Setting a high fill factor provides benefits,

such as lower storage requirements, shorter scans, less I/O operations to access the data and more content stored in the data cache. Nevertheless, there are also some disadvantages, such as slower inserts and data splits, suffering concurrency because more data is unavailable to other users when a page is locked, less space to expand for variable-length attributes and more complicated reorganization.

Page size (or block size) is a parameter used to specify the appropriate size of data page to ensure efficient data storage and I/O. Usually, page size is limited to a number of choices, such as 4k, 8k, 16k etc. To determine which page size to use, one should take into account the length of each table row as well as any page or row preambles. After that, he should also consider the desired amount of free space in each page. Finally, one might want to choose the page size that saves the most space. For instance, if the record size together with the equivalent free space is 2500 bytes, the 4K page would hold one record per page, while the 8K page size would hold three. Specifying a high page size means more data in the buffer pool and, therefore, better hit ratio and less I/O operations. However, it also means longer I/O when a miss does occur and inadequate concurrency because more data per page gets locked.

6.1.3 Application Tuning

Application tuning involves developing efficient SQL code to facilitate optimization procedure and it is usually the main cause of performance problems. However, in the case of running TPC-H the SQL queries are already defined, therefore we will not discuss further this tuning area.

6.2 Performance Tuning Issues for Ad-hoc Decision Support Workloads

There has been considerable amount of research on the topic of automatically characterising a database workload as OLTP or DSS [8] [9] [10]. Determining whether a specific workload is of OLTP or DSS nature makes a big difference to the database administrators (DBAs). In fact, these database experts consider the workload type a key criterion for their tuning decisions. They apply rules-of-thumb tuning strategies to handle each workload. Therefore, they must recognise the significant shifts in the workload and reconfigure the system accordingly in order to maintain acceptable levels of performance as we cannot optimize for both workloads. Another reason why automatic characterization of a database workload is so important is to assist the development of self-tuning database systems that would take into account the workload type and reconfigure themselves accordingly [11] [12]. Some database ven-

dors provide recommendations for tuning according to the workload type [13] [14] [15] [16].

We are interested in the characteristics of ad-hoc DSS workloads and the tuning parameters that affect them. DSS queries most of 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.

First of all, the complexity of DSS queries makes efficient query optimization of vital importance. We will discuss this in detail in chapter 7. What is more, since optimizing complex queries is expensive in terms of time and resources, it would be very beneficial to keep many query plans in the procedure cache. However, by nature DSS queries are hardly ever repeated and we are examining ad-hoc ones. Therefore, we will not explore further this option.

Secondly, the fact that these queries deal with large amounts of data within scans, sorts and joins dictates that 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.

Thirdly, 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.

Chapter 7

The SQL Server and MySQL Query Optimizers

As mentioned in the previous chapter, the performance of DSS workloads is highly influenced by the query optimization process. For that reason, we are going to examine this database system component in detail and find the parameters that affect its behavior in each one of the two systems.

7.1 Overview of the Main Components of an RDBMS

As illustrated in [6], the main components of a relational database management system are as in Figure 7.1. In order to execute a query, we first have to establish a connection between the client and the database through the *communications manager*. The *process manager* will then decide whether the system has enough resources available for the execution. Next, the query is appropriately rewritten and optimized by the *query processor*, compiled into an internal query plan featuring specific operators and executed accordingly. Finally, the *transactional storage manager* oversees the data access during execution, by ensuring that the ACID properties are met and the data is accessed following basic structures, such as tables and indexes.

7.2 Architecture of a Query Processor

The *SQL parser* is responsible for query parsing and authorization. During parsing, the SQL statement is checked for syntax correctness and all the table names and attributes are resolved and checked for existence. Then, the query is converted into the internal format used by the optimizer.

The *query rewriter* is responsible for simplifying and rewriting the query into a

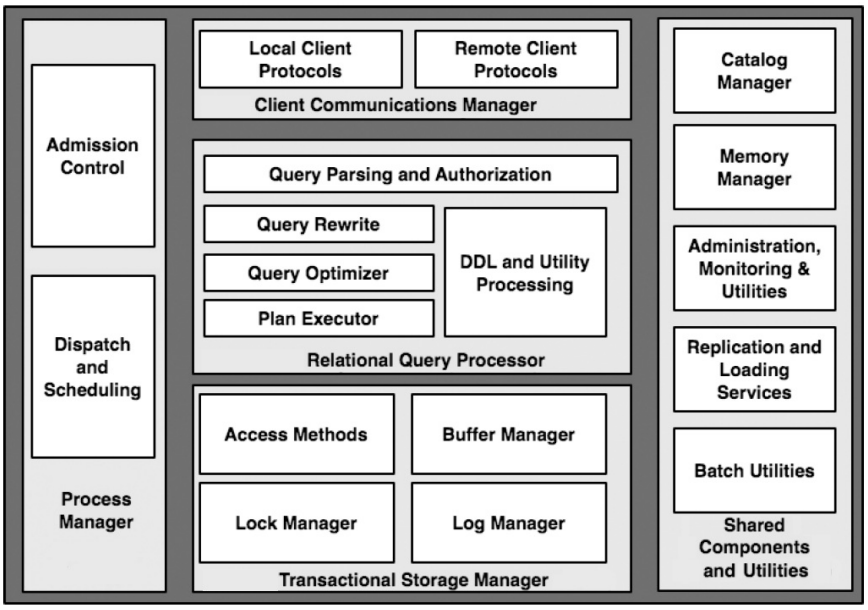


Figure 7.1: Main components of an RDBMS.

common format without changing its semantics, operating on its internal representation. Among others, this step handles view expansion, meaning that it rewrites the query replacing view references with the corresponding tables and attributes. It also rewrites the predicates in the simplest way possible and eliminates redudant table joins. Finally, since most optimizers operate on individual SELECT-FROM-WHERE blocks and do not optimize across blocks, it is the responsibility of the rewriter to flatten nested queries when possible in order to maximize optimization opportunities.

The *query optimizer* transforms the internal query representation into an efficient query plan. A *query plan* can be thought as a dataflow diagram that pipes table data through a tree of *physical operators*, such as sorts, joins and loops. In other words, it is a physical operator tree representing the sequence of operations for the execution of the query.

The *plan executor* receives the query plan from the query optimizer and executes the physical operators. In other words, it compiles the logic into an executable program.

7.3 Overview of the Query Optimization Process

The goal of query optimization is to generate an efficient query plan for the execution of the given SQL statement. The complexity lies on the fact that there are more than one query plans for each SQL query. Each query can have many equivalent algebraic representations and each algebraic representation can be implemented

using different physical operators with different computational and I/O costs. The optimizer is expected to choose the most efficient combination of physical operators for a given query.

Defining query plan efficiency is another important issue. Some plans may require fewer resources than others but other plans may run faster. The goal is to discover the plan that has the least cost and/or runs in the least amount of time. Sometimes one has to sacrifice resource usage for execution time and vice versa, such as running an application on a small platform with low resource availability or, in contrast, when there is a need for higher execution throughput.

The first query optimizers were developed for early database systems, such as System /R [21] and INGRES [22]. The main concepts introduced by these components have been incorporated in many commercial optimizers and continue to be remarkably relevant. The traditional query optimization in System /R applies to Select-Project-Join (SPJ) queries, and requires the definition of a search space, a cost estimation technique and an enumeration algorithm [7]. However, this relational model was extended to include object-oriented and distributed database systems, giving birth to a new class of extensible optimizers such as the Volcano optimizer [23] that formed the basis for the Microsoft SQL Server optimizer.

The *search space* is the theoretical set of all possible physical query plans for a given query. Each system has its own toolset of operators and algebraic transformations that help it create the search space for each query. There is no system that considers all possible operators and transformations; thus each system considers only a subset of the total possible search space. For instance, in System /R acceptable query plans corresponded to a linear sequence of join operations, such as Join(Join(Join(A,B),C),D).

The *cost estimation technique* assigns an approximate cost in terms of resources or/and time to any partial or complete query plan. It also determines the estimated size of the output data stream. In order to do so, it collects statistics on tables and indexes, such as the number of pages in a table or index and the number of distinct values per column. Subsequently, it applies a number of formulas for estimating the predicate selectivity, the size of the output data as well as the CPU and I/O costs for each physical operator. Then, the total cost for the query plan can be obtained by combining the costs of each of the operator nodes in the tree. Not all systems adopt the same cost estimation technique which means that, even if they explore the same search space they do not select the same query plan in the end.

The *enumeration algorithm* has to explore the search space of cost-estimated query plans in quest of the most inexpensive one. That is a notoriously difficult search

problem and many computer science techniques have been employed to solve it efficiently. Database systems differ on implementing this algorithm too. The System /R approach made use of dynamic programming and interesting orders. *Dynamic programming* refers to solving complex problems by breaking them down into simpler steps. In this case, the optimizer assumes that in order to obtain an optimal query plan for a SPJ query consisting of k joins, it suffices to consider only the optimal sub-plans consisting of $(k-1)$ joins. The method of *interesting orders* ensures that the system collect information from sub-trees to help make decisions about a higher point in the tree. For instance, should we have a predicate like $R1.a = R2.a = R3.a$, it is smart to use a sort-merge join on the sub-query R1, R2 so as to get the result of the join sorted on a and thus reduce the cost of the join with R3. Therefore, even if it is cheaper to use an other type of join (such as nested-loop join) for the sub-query, it is more beneficial in total to go with the sort-merge one.

The idea of interesting orders was generalized to *physical properties* and used extensively in modern optimizers. For example, consider a simple query of the form $select col1, col2, max(col3) from table1 group by col1, col2$. Now, if the columns (col1, col2) make up a unique key on table Table1, then it is not necessary to do grouping at all: each group has exactly one row. The max() of a set of size one is the element itself. So it is possible to remove the group by from the query completely. Applying this procedure constitutes using a physical property.

Today, there are four primary types of query optimization: cost-based, heuristic, semantic, and parametric optimization. None of these techniques can guarantee an optimal query plan. Instead, they constitute different approaches for the common goal of generating an efficient query plan that guarantees correct results.

7.3.1 Cost-based Optimization

Cost-based optimizers follow the basic philosophy of System /R optimization. Their goal is to utilize indexes and statistics gathered from past queries to predict the cost for each possible query plan and then choose the less expensive one. Systems such as Microsoft SQL Server and Oracle use cost-based optimizers.

7.3.2 Heuristic Optimization

As the hardware improved, differences in execution time per query plan tend to diminish. In fact, most query plans have been shown to execute with approximately the same cost. This realization has led some database system implementers to adopt a query optimizer that focuses on optimizing the query using some well-known good rules (called heuristics) or practices for query optimization. Thus, the goal of *heuristic optimization* is to apply rules that ensure "good" practices for

query execution. Systems typically use heuristic optimization as a means of avoiding the really bad plans rather than as a primary means of optimization, resulting in optimization technique hybrids. Systems that use purely heuristic optimizers include POSTGRES and various academic variants. One example of hybrid implementation is that of the query optimizer of MySQL.

7.3.3 Parametric Optimization

Parametric optimization combines the application of heuristic methods with cost-based optimization [24]. The resulting query optimizer provides a means of producing a smaller set of effective query plans from which cost can be estimated, and thus the lowest-cost plan of the set can be executed. Query plan generation is created using a random algorithm, called sipR. This permits systems that utilize parametric query optimization to choose query plans that can include the uncertainty of parameter changes (such as buffer sizes) to choose optimal plans either formed on the fly or from storage. The concept of parametric optimization suggests that the use of dynamic programming algorithms may even not be needed. It was documented that for small queries, dynamic programming is superior to randomized algorithms, whereas for large queries the opposite holds true [24].

7.3.4 Semantic Optimization

The goal of *semantic optimization* is to form query execution plans based on the semantics of the database. It is assumed that the optimizer has a basic understanding of the database schema. When a query is submitted, the optimizer uses its knowledge of the system constraints to simplify or to ignore a particular query if it is guaranteed to return an empty result set [38]. Though not yet implemented in any commercial database systems as the primary optimization technique, semantic optimization is currently the focus of considerable research [39] [40] [42].

7.4 The Microsoft SQL Server Query Optimizer

7.4.1 Query Optimization Process

According to Microsoft SQL Server textbook [26], the SQL Server query optimizer is designed around a classic cost-based optimizer, based on the Volcano/Cascades optimizer generator [23] [27]. Query optimization in SQL Server 2008 comprises six steps: *(i)* query simplification; *(ii)* search for trivial plan; *(iii)* statistics creation or update; *(iv)* exploration/implementation phases; *(v)* conversion to execution plan; and *(vi)* query plan caching. The first step is query simplification. The optimizer receives the parsed query and breaks it down to operations that need to be com-

pleted. It stores that information in a syntactically normalized format. Then, it checks for obvious contradictions in the predicates (i.e. *where col1* > 0 *and col1* < -3); in case of a contradiction, the optimization procedure stops at this point and the user receives an empty result set.

If applicable, the optimizer now proceeds with the generation of a trivial plan. In order to speed up simple queries (i.e. OLTP workloads) that do not correspond to many query tree alternatives, the optimizer quickly generates a trivial plan for them and concludes the optimization procedure. The reasoning behind this implementation decision is that the time spent choosing the best among these few query plans could be longer than the time spent executing the worse of them.

SQL Server 2008 also gives the user the option to force a specific query plan. This option is considered at this step, thus stopping further optimization.

The next step is the creation or update of the relevant statistics. At this point, the optimizer creates or updates statistics on the columns that appear at the predicates. When statistics are created, a number of table or index rows is sampled to collect statistical information. Statistics are updated when the number of rows that have been updated is comparable to a fraction of the total rows. The statistics of SQL Server contain: *(i)* an histogram of the data distribution per column participating in the statistics collection; *(ii)* header information with information such as the number of rows sampled during the statistics creation; *(iii)* trie trees with data distribution information for string columns; and *(iv)* density information, which is the average number of rows returned per unique value.

At the core of the optimization procedure, we have the exploration/implementation phases. The optimizer starts from the initial query tree and transforms it into many equivalent logical trees, using specific exploration rules. Consequently, it generates a number of physical trees from each logical tree, using specific implementation rules. Then, it calculates the cost of each physical tree and temporarily stores them in a special memory component called *the Memo*. Finally, when all query plans have been generated and their cost has been calculated, the optimizer chooses the cheapest plan in the Memo.

In order to speed up less complex queries, the optimizer separates exploration and implementation rules based on cost and how likely they are to be useful. Therefore, it executes this step in three phases, each time finding the cheapest plan after applying a set of rules. If at the end of the first or the second phase the estimated cost of the cheapest plan found is comparable to the time spent in optimization so far, this plan is adopted and the optimization procedure is concluded. One of the most important priorities when applying these rules is to make use of indexes whenever

it is possible, that is to say whenever a predicate matches a column index.

SQL Server largely bases its costing model on the number of I/Os. In order to calculate the cost of each query plan, the optimizer estimates the selectivity of each predicate by retrieving the sampled information from the statistics and scaling it to the current table size. Then, based on the selectivity and the average row length, it calculates the number of data pages that will have to be read from the disk and, thus, the number of I/Os. In order to estimate this as realistically as possible, the optimizer makes assumptions on the number of sequential and random I/Os that will be necessary, as well as the number of pages that will be cached and read multiple times.

Since the query plan search is not exhaustive and the costing model is not ideal, it is not feasible for the optimizer to always find the optimal plan. Therefore, the user is given the option of overriding the optimizer's decisions by submitting an optimization hint. If the user has chosen to submit such a hint, the optimizer discards all rules contradictory to the hint and only generates query plans that comply with it. However, it still looks for the cheapest plan among them.

At the end of the optimization procedure, the cheapest plan that has been found is converted into an executable version. The query is then saved in procedure cache together with the selected execution plan. Simple queries and corresponding plans are saved in a parameterized format, in order to be re-used when similar queries with different comparison values in their predicates come up. This option cannot be used for highly complicated DSS queries where even the selectivity of each predicate makes a difference in the resulting query plan. Some special cases are the stored procedures whose execution plan is cached before being called by the user and the bulk inserts whose execution plan is never cached.

7.4.2 Controlling the SQL Server Optimizer

There are two ways to directly override the query optimizer: plan forcing and plan hinting. Plan forcing means providing the optimizer with a pre-prepared execution plan to follow; thus, it does not make sense for ad-hoc DSS queries. Plan hinting, on the other hand, includes table hints that allow the user to force index selection and join hints that allow the user to specify the type of join strategy used.

Another way to influence the optimization procedure as an experiment would be creating and updating statistics manually instead of automatically, thus the optimizer would not always have the most up-to-date information for assigning a cost to each operator. Finally, we could modify the estimated number of I/Os and the available space for sorting and storing intermediate results which the optimizer takes

into account, by changing the page fill factor or the buffer pool size.

7.5 The MySQL Query Optimizer

7.5.1 Query Optimization Process

Our presentation of the principles underlying the MySQL query optimizer relies largely on MySQL textbook [25]. In one sentence, the MySQL optimizer is a hybrid, combining a cost-based and a heuristic optimizer.

The most important heuristic in MySQL is that it tries to eliminate as many rows as possible as soon as possible. This might sound a little counter-intuitive, since the goal of a query is to find rows, not to reject them. However, MySQL developers recognise that joins are the most expensive and time consuming of all of the relational operators and therefore it would be better to perform them on a subset of the original tables or indexes. Therefore, the optimizer identifies the relational algebra operations to be computed for the execution of the query, and it uses a Restrict-Project-Join strategy.

First, it performs the Restrictions by evaluating the predicates in the Where clause. Thus, it reduces the number of rows to work with. During this step, it follows another important heuristic: it tries to use indexes whenever possible. Then, it performs the Projections that appear in the Select clause, in order to reduce the number of columns in the resulting rows. Finally, it performs the Joins in the From clause as well as the sorts in the Group by and Order by clauses. At this last stage, the query is optimized by eliminating any known-bad conditions and finding the cheapest way to compute the joins. In order to do so, it uses a I/O costing method based on statistics as well as a search method, both very similar in principle to the SQL Server's optimizer.

7.5.2 Controlling the MySQL Optimizer

As explained, the optimizer tries to use indexes whenever possible. Nevertheless, the user has some control over access method selection with the commands 'use index', 'force index' and 'ignore index'.

It is also possible to override the optimizer decisions by forcing a particular join order with the command 'straight_join'. However, the optimizer is using the restriction heuristic in this case as well: it attempts to order the tables so that the first table is the one from which the smallest number of rows will be chosen. This is hard to guess unless we have a very good knowledge of the data distribution; therefore, forcing a particular join order rarely helps performance.

In addition, there are two system variables that affect join optimization: optimizer_prune_level and optimizer_search_depth. The first variable tells the optimizer to skip certain plans based on estimates of the number of rows accessed for each table. MySQL developers believe that this is an 'educated guess' that rarely misses optimal plans, while it speeds up the optimization procedure. The second variable tells the optimizer how far into the 'future' of each incomplete plan it should look to evaluate whether it should be expanded further. The smaller the value of this variable, the smaller the complexity of the search algorithm and the smaller the optimization time. Once again, developers advise that this is set to a low value, as they believe that, in most of the cases, the performance benefits of executing an optimal plan do not compensate for the time it takes to find it.

Finally, just like in SQL Server, the cost-based part of the optimization process relies on data statistics. Therefore, their creation and update influences the optimizer behavior.

Chapter 8

Test Results and Analysis

In chapter 6, we explained that the performance of ad-hoc decision-support workloads can be enhanced by tuning a number of system and database parameters. We also demonstrated the importance of query optimization, thus exploring the parameters that affect each query optimizer in chapter 7. In this chapter, we are going to perform a series of experiments: first, we will run some full TPC-H tests while varying the tuning parameters from chapter 6 in order to compare the database systems overall price/performance ratio; then, we will experiment with the parameters from chapter 7 and observe the behavior of the two query optimizers by examining the generated query plans.

8.1 Full TPC-H Tests

8.1.1 Parameters Varied in the Experiments

As mentioned in chapter 6, the following tuning parameters influence the performance of ad-hoc decision-support queries: *(i)* buffer pool size; *(ii)* sort buffer size; *(iii)* fill factor; *(iv)* page size; and *(v)* enabling of intra-partition parallelism. Since we are running the tests on a single-core machine, we cannot evaluate the effect of the last parameter; however, we are going to run the full TPC-H test for various values of the rest of the parameters. Tables 8.1 and 8.2 display the names and values range of these parameters in Microsoft SQL Server 2008 and in MySQL 5.1.

MS SQL Server 2008				
	Min	Max	Default	Parameter Name
Total Server Memory	0	2 PB	0	min server memory
	16 MB	2 PB	2 PB	max server memory
Fill Factor	1	100	0 (=100)	fill factor
Page Size	n/a	n/a	8K	n/a

Table 8.1: Tuning parameters affecting DSS query performance in SQL Server 2008.

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
Page Size	8 KB	64 KB	16 KB	univ_page_size

Table 8.2: Tuning parameters affecting DSS query performance in MySQL 5.1.

In SQL Server, the user can only set the total size of memory that the DBMS can use, by setting its minimum and maximum values. Then, the DBMS automatically allocates the available memory to the various caches, according to the workload needs. MySQL, on the other hand, allows the user to set a specific size for the buffer pool and the sort buffer. Furthermore, while SQL Server operates with a fixed page size of 8 KB, in MySQL the user can set 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.

In the next pages, we are going to present the results of running the full TPC-H test in each of the two systems, under various configurations. 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 898$ for SQL Server 2008 and 0$ for MySQL 5.1.

8.1.2 Full TPC-H Tests in SQL Server

Tables 8.3 and 8.4 show the test results for several configurations of the two systems, varying the parameters in Table 8.1.

MS SQL Server 2008							
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%
load test	46min	20min	19min	17min	16min	16min	36min
performance 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$

Table 8.3: TPC-H full test results for increasing memory size in MS SQL Server 2008.

In the first seven tests with SQL Server in Table 8.3, 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. Specifically, we set both min server memory and max server memory parameters at the same value, so that we achieve a fixed memory size. As we change the value from 16 to 768 MB, the performance improves significantly; in fact, the performance difference between two consequent values is impressive in

MS SQL Server 2008					
total server memory fill factor	128 MB 40%	128 MB 60%	128 MB 80%	128 MB 90%	128 MB 100%
load test performance test	27min 2h2min	22min 1h9min	20min 1h3min	19min 1h	19min 59min
Power@1GB	24.86qph	117.41qph	119.42qph	119.60qph	120.01qph
Throughput@1GB	48.13qph	54.64qph	66.83qph	68.04qph	69.89qph
QphH@1GB	34.59qph	80.10qph	89.34qph	90.20qph	91.58qph
Price-per-QphH@1GB	40.42$	17.45$	15.65$	15.49$	15.23$

Table 8.4: TPC-H full test results for increasing fill factor in MS SQL Server 2008.

the beginning but, as we move towards higher memory values, the effect is not that dramatic. The system ends up reaching its full potential around 512 MB. After that, 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 and store intermediate results. Finally, when we run the test with 1024 MB of server memory, the performance suffered severely compared to the previous value of 768 MB. The reason for that is that in this case the server uses up all available memory in the system, thus causing the operating system to perform poorly.

Subsequently, we experimented with 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. The results of these tests are displayed in Table 8.4. Indeed, this time varying the fill factor did have a major impact on performance. The explanation for this is that a high memory size permitted 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.

In the five tests with 128 MB and different fill factors in Table 8.4, we observe once again that the performance is improving rapidly in the beginning and stabilizing as the value increases. Our conclusion is that the server more or less reaches its potential around 90%, even though the performance deterioration is not that significant for 80% too. Just like in the case of memory size, the difference in performance is a result of the server's ability to keep more data pages in cache, thus avoiding

expensive I/Os, and storing larger intermediate results, thus being able to choose query plans that require such space and might run faster. This last property will be verified in one of the following experiments. 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 due to page splits and bad concurrency because more data gets locked with each page; 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.

As a final note, examining the performance in Tables 8.3 and 8.4 in terms of the TPC-H price/performance metric, we can claim that increasing the memory size from 16MB to 512MB is equivalent to a 85.37% cost reduction, while increasing the fill factor from 40% to 100% is equivalent to a 62.32% cost reduction. Therefore, it looks like specialized tuning for DSS queries can help us save a lot.

8.1.3 Full TPC-H Tests in MySQL

MySQL 5.1							
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
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$

Table 8.5: TPC-H full test results for increasing memory size in MySQl 5.1.

MySQL 5.1				
buffer pool size	96 MB	96 MB	96 MB	96 MB
sort buffer size	32 MB	32 MB	32 MB	32 MB
page size	8 KB	16 KB	32 KB	64 KB
load test	20min	18min	17min	17min
performance test	1h13min	59min	52min	50min
Power@1GB	114.67qph	122.58qph	141.73qph	144.88qph
Throughput@1GB	55.59qph	69.66qph	79.57qph	82.58qph
QphH@1GB	79.84qph	92.41qph	106.20qph	109.38qph
Price-per-QphH@1GB	6.26$	5.41$	4.71$	4.57$

Table 8.6: TPC-H full test results for increasing page size in MySQl 5.1.

Tables 8.5 and 8.6 hold the results for MySQL. We experimented with different memory sizes, while keeping the page size at 8 KB and with various page sizes while

keeping the sum of the buffer pool and sort buffer sizes at 128 MB. We chose the page size of 8 KB for the memory tests in order to be able to compare these results with the ones from SQL Server that features a fixed page size of 8 KB. Following the same logic, we chose a total size of 128 MB of memory for the page size tests. At last, we chose to allocate three quarters of the total memory to the buffer pool and the rest to the sort buffer, following the tuning advice in MySQL documentation.

The results of the MySQL tests in Tables 8.5 and 8.6 illustrate the same patterns in performance difference for the same reasons as in the SQL Server tests: the more space the server has for caching data pages and intermediate results from sortings and other operations, the better the performance. Once again, the performance improves signiicantly between lower values and the effect becomes less visible as the values increase. Also, for memory equal to the total available system memory, the performance suffers. Like in SQL Server, tuning more appropriately leads to impressive reductions on the cost: 83.22% price/performance metric difference between the test with 16MB of total memory and that with 512MB; and, 27.00% difference between page size of 8KB and 64KB.

One could argue that a higher page size should not improve performance, as in the best case scenario it means the same amount of data in the buffer pool only differently organized and in the worse case scenario it means more irrelevant data occupying space in the buffer pool because they were fetched together with relevant ones leaving no space for new data, and more data getting locked with each page by concurrent users. This last part we cannot testify upon, since we are running the tests on a single-core machine. However, as far as the amount of relevant data brought into cache with every page is concerned, we remind here that 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 space goes wasted in non-data preambles. Finally, depending on the size of the records in each table, 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 between the test with page size 8 KB and the one with 16 KB.

8.1.4 Comparison of SQL Server and MySQL Overall Performance

It is important to note here that the test configurations in the two systems are not equivalent. In fact, total server memory in SQL Server is used for all memory needs,

the most important of which are the buffer pool, the sort buffer and the procedure cache. To eliminate the memory consumed by the procedure cache, we configured the query optimizers not to cache the query plans, since they are not re-used anyway as we are dealing with ad-hoc highly complex DSS queries. Therefore, we can claim that, by allocating total server memory size in SQL Server equal to the sum of the buffer pool and the sort buffer sizes in MySQL, we have configured the memories as fairly as possible. Another issue is that SQL Server operates on a fixed page size of 8 KB but gives the user the right to change the page fill factor, while in MySQL the user has a choice of page sizes but the fill factor of every page is controlled dynamically by the server leading to pages with a fill factor between 70% and 95%. By running tests with fill factor 90% in SQL Server and page size 8 KB in MySQl, we made the tests more or less comparable.

That being said, even if no configuration is equal to another, we can still draw some conclusions are to how the two systems compare. Examining the first seven tests in the results tables, we can see that, for the same page size, more or less 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 control of the fill factor in MySQL cannot compete with a user-defined high fill factor in SQL Server, or that there are some more tuning parameters (other than the major ones that we mentioned in chapter 6 and which we 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, as we are dealing with highly complex DSS queries. This conclusion is strengthened by the fact that 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. We will explore the query optimizers performance some more in the next experiments.

These results lead us also to understand that, 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 favours MySQL. The additional 898$ required to purchase a SQL Server 2008 license do not seem to be worthy 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, for a global corporation with billions of clients and suppliers, this performance difference can end up cost more than a SQL Server license.

8.2 Experiments with the Query Optimizers

Back to parameters that affect the performance of DSS workloads, in chapter 6 we also demonstrated the importance of query optimization. Various parameters affect query optimization in each one of the DBMSs, as presented in chapter 7. In a nutshell, both optimizers would be influenced by the quality of available updates and both systems allow optimizer hints. In addition, SQL Server optimizer seems to be sensitive to changes in cache data potential, while the user can customize the exhaustiveness of the query plan search of the MySQL optimizer. We will experiment with these factors and observe the influence they exert on the optimization of TPC-H queries, by examining the query plans that are generated in each case as well as the total difference in performance. First, we are going to show the effect of buffer pool size and fill factor on the query plan choice of SQL Server; then, we will proceed with some experiments with MySQL's optimizer_search_depth and optimizer_prune_level.

Let us note here that our initial idea was to experiment with the optimizers when there are no statistics avaiable, thus intending to observe the optimizer's behavior when it does not have realistic selectivity information on each predicate. To this goal, we proceeded in running tests in SQL Server while not collecting statistics during the load test and having turned off their automatic creation and update. To our surprise, however, the two tests took the same time to run and the generated execution plans were identical! Taking a closer look on statistics collection in SQL Server, we found out that index creation automatically triggers statistics collection on them, and these statistics cannot be dropped while the index exists. Therefore, since we decided to use indexes for the tests, it is impossible to avoid statistics collection; plus, these statistics remain almost accurate even though they are not updated, since the workload contains very few updates. Meanwhile, MySQL does not allow the user to stop creating and updating statistics.

When statistics are created in the two systems however there is an important difference: MySQL samples only 8 pages of table data, by default; SQL Server samples a minimum of 8MB of data (unless the entire table is smaller than that) and processes the 200 most represented unique values. This means that, even for a page size of 64KB, MySQL would only be sampling 0.5MB as opposed to SQL Server's 8MB. Therefore, the statistics in SQL Server are more accurate and can lead to more precise costing of query plans and better final choices.

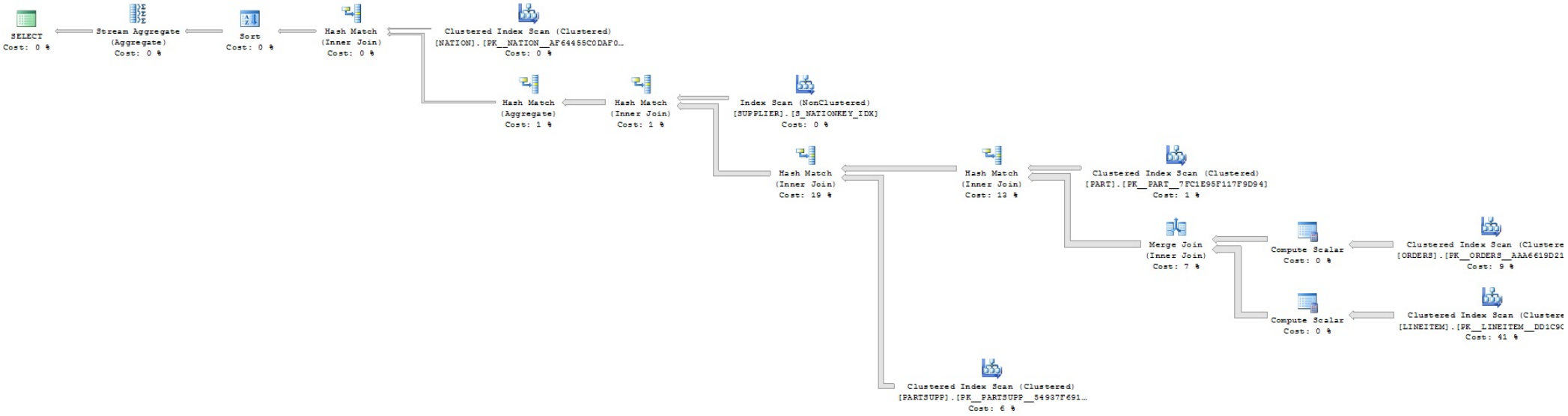


Figure 8.1: Execution plan for query 9 for total memory size 512 MB and fill factor 90%.

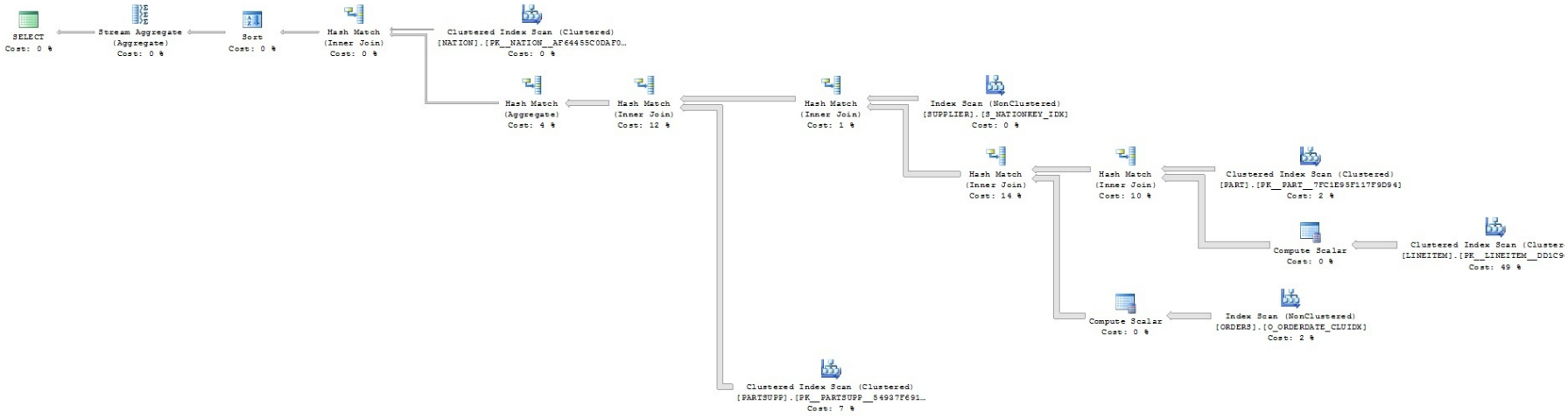
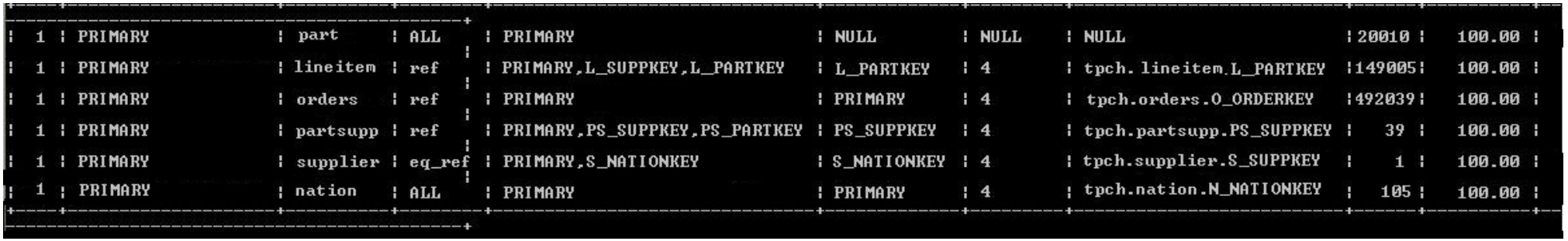


Figure 8.2: Execution plan for query 9 for total memory size 16 MB and fill factor 40%.



1	PRIMARY	part	ALL	PRIMARY	NULL	NULL	NULL	20010	100.00	
1	PRIMARY	lineitem	ref	PRIMARY,L_SUPPKEY,L_PARTKEY	L_PARTKEY	4	tpch.lineitem.L_PARTKEY	149005	100.00	
1	PRIMARY	orders	ref	PRIMARY	PRIMARY	4	tpch.orders.O_ORDERKEY	492039	100.00	
1	PRIMARY	partsupp	ref	PRIMARY,PS_SUPPKEY,PS_PARTKEY	PS_SUPPKEY	4	tpch.partsupp.PS_SUPPKEY	39	100.00	
1	PRIMARY	supplier	eq_ref	PRIMARY,S_NATIONKEY	S_NATIONKEY	4	tpch.supplier.S_SUPPKEY	1	100.00	
1	PRIMARY	nation	ALL	PRIMARY	PRIMARY	4	tpch.nation.N_NATIONKEY	105	100.00	

Figure 8.3: Execution plan for query 9 in MySQL.

```
select
        nation,
        o_year,
        sum(amount) as sum_profit
from
        (
                select
                        n_name as nation,
                        extract(year from o_orderdate) as o_year,
                        l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
                from
                        part,
                        supplier,
                        lineitem,
                        partsupp,
                        orders,
                        nation
                where
                        s_suppkey = l_suppkey
                        and ps_suppkey = l_suppkey
                        and ps_partkey = l_partkey
                        and p_partkey = l_partkey
                        and o_orderkey = l_orderkey
                        and s_nationkey = n_nationkey
                        and p_name like '%dark%'
        ) as profit
group by
        nation,
        o_year
order by
        nation,
        o_year desc;
```

Figure 8.4: Query 9 text.

8.2.1 Experiments with SQL Server

During our SQL Server tests with different memory sizes and fill factors, we found that some of the queries were executed using different query plans. The objective of the following two experiments is to explain why this occured and how it influenced performance.

Figures 8.1 and 8.2 present the execution plans generated for the same query (query 9) in SQL Server with high total memory size and fill factor settings, and with low total memory size and fill factor settings, respectively. Query 9 is a join-bound query and with this experiment we intend to show how these two parameters influence the implementation of joins. You can see the query text in Figure 8.4. Reading the plans from right to left, we see that in Figure 8.1 the join order is

$(((((Orders \& Lineitem) \& Part) \& Partsupp) \& Supplier) \& Nation)$

while in Figure 8.2 the join order is

$(((((Part \& Lineitem) \& Orders) \& Supplier) \& Partsupp) \& Nation)$

The first two joins in each plan involve the same three tables: Lineitem, Orders and Part. With scale factor 1, Lineitem has 6.000.000 rows, Orders 1.500.000 and Part 200.000. Therefore, joining Orders with Lineitem requires a much larger buffer than joining Part with Lineitem for any kind of join algorithm, not only for the join operation itself but also for storing the intermediate result. However, it may be worth it in terms of performance, as both Orders and Lineitem have indexes

on the column Orderkey so they can be joined with a merge join, which is less expensive than a hash join. Hence, we observe that the low settings plan chose a slower solution because it lacked the means to implement the faster one.

Moving to the next two joins, we see that once again they involve the same tables (Partsupp and Supplier) but in reverse order. Partsupp has 800.000 rows while Supplier has only 10.000. So, once again because of limited available memory, the low settings test chooses to implement the join with Supplier first, thus avoiding to store large intermediate results.

Query 9 ran in 35 seconds with high settings and in 57 seconds with low settings. This proves the superiority of the first plan, even though it could not be implemented in the second case due to space limitations. To verify our theory, we forced the first plan's join order under low settings; the query timed out at 5 minutes without returning results. Thus, the available memory was indeed too small to use this plan. Finally, we forced the second query plan's join order under high settings and the query ran in 49 seconds; hence, the second plan is indeed slower. An extra conclusion that we can draw from this experiment is that, even for highly complicated queries, SQL Server's query optimizer should be able to recognise the optimal join order and apply it if there are adequate resources. Therefore, forcing the query's join order does not seem likely to enhance performance; it is rather a tool developed for experiments like this one.

Finally, Figure 8.4 shows the execution plan generated by MySQL. This plan does not change, regardless the buffer pool size and it follows the 'safe path', that is to say it is executable even in a small pool. This leads us to believe that the MySQL optimizer is not memory-sensitive and, therefore, cannot take advantage of the potential benefits of an increased memory as far as possible query plans are concerned.

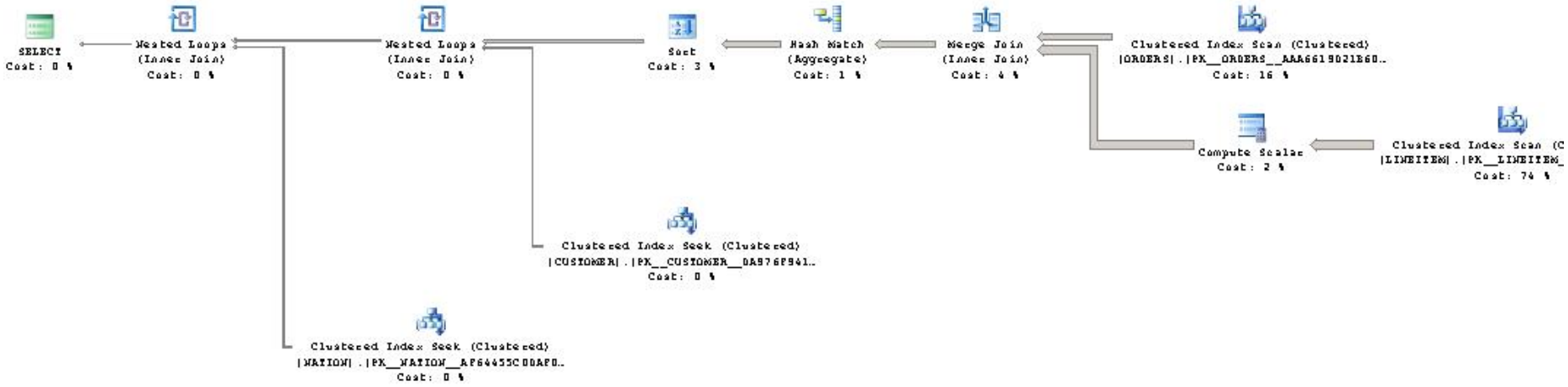


Figure 8.5: Execution plan for query 10 for total memory size 512 MB and fill factor 90%.

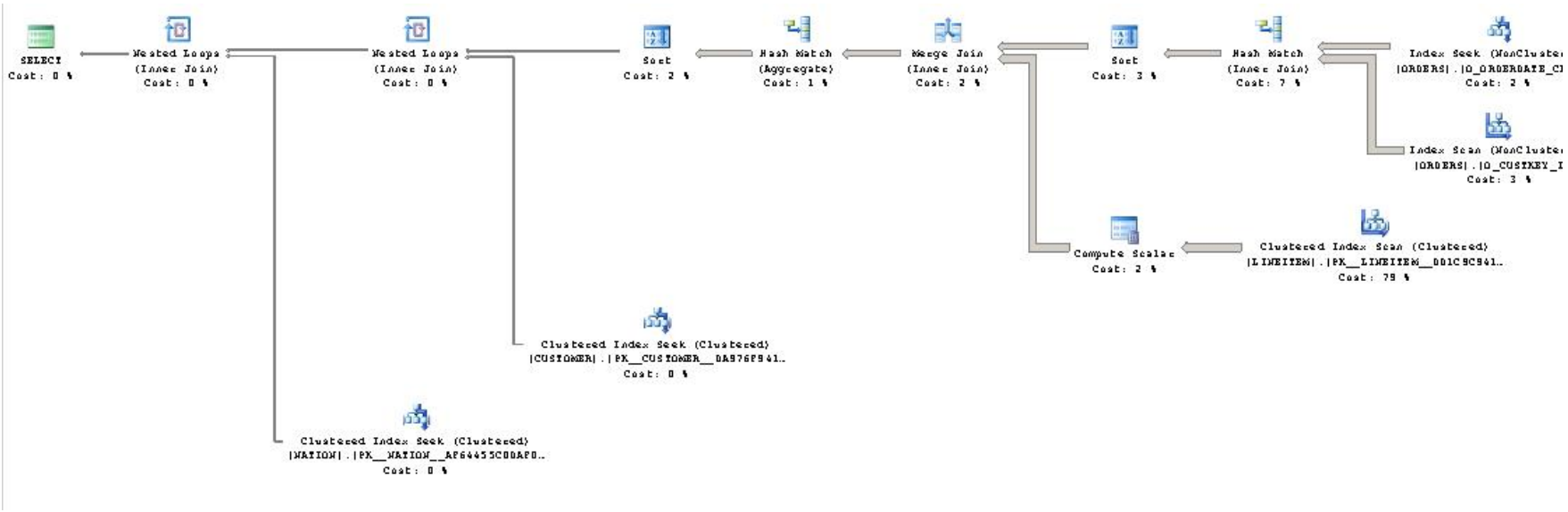


Figure 8.6: Execution plan for query 10 for total memory size 16 MB and fill factor 40%.

```
select
        sum(l_extendedprice) / 7.0 as avg_yearly
from
        lineitem,
        part
where
        p_partkey = l_partkey
        and p_brand = 'Brand#52'
        and p_container = 'MED DRUM'
        and l_quantity < (
                select
                        0.2 * avg(l_quantity)
                from
                        lineitem
                where
                        l_partkey = p_partkey
        );
```

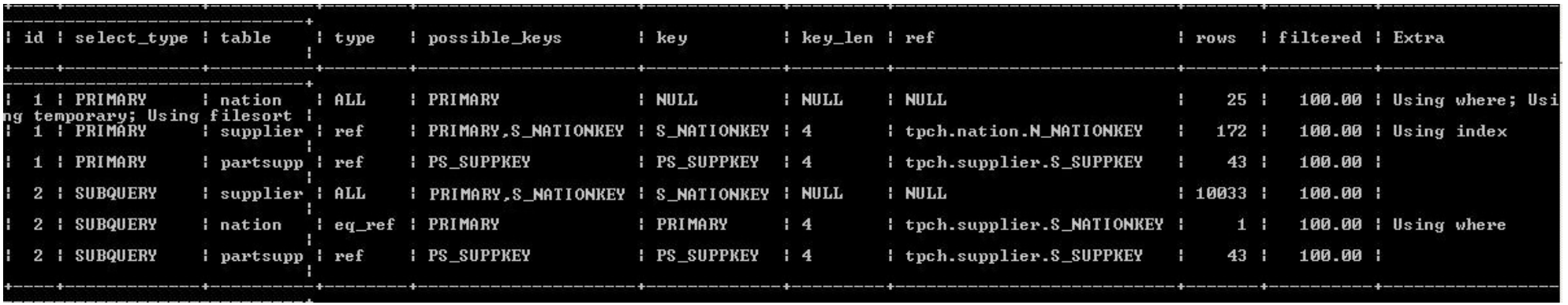
Figure 8.7: Query 10 text.

In Figures 8.5 and 8.6, we have the execution plans generated for query 10 running in SQL Server first under high total memory size and fill factor settings, and then under low total memory size and fill factor settings. The query text is shown in Figure 8.7. The purpose of the experiment is to understand the optimizer's access path choices when running scan-bound queries under different values of memory size and fill factor.

Examining the execution plans, we see that their only difference is the method with which they chose to access the table Orders. The part that is different is located at the upper-right part of the figures. In order to enter the merge join with Lineitem, the scan results from table Orders have to be sorted on the Orderkey column; meanwhile, the scan has to evaluate the date predicates and return columns Orderkey and Custkey. In cases like this, whenever possible, SQL Server evaluates the predicate within the scan. Indeed, it did so in the case of high settings. However, this predicate is too expensive to evaluate in terms of memory resources; therefore, in the case of low settings, the optimizer chose to use a separate index seek to find the rows that qualify the date predicates and then use another index in an index scan as a hash join probe. Of course, after that it had to sort the results to prepare them for the merge join with Lineitem, all in all resulting in worse performance. Indeed, the query took 43 seconds to execute in low settings and only 29 in high settings.

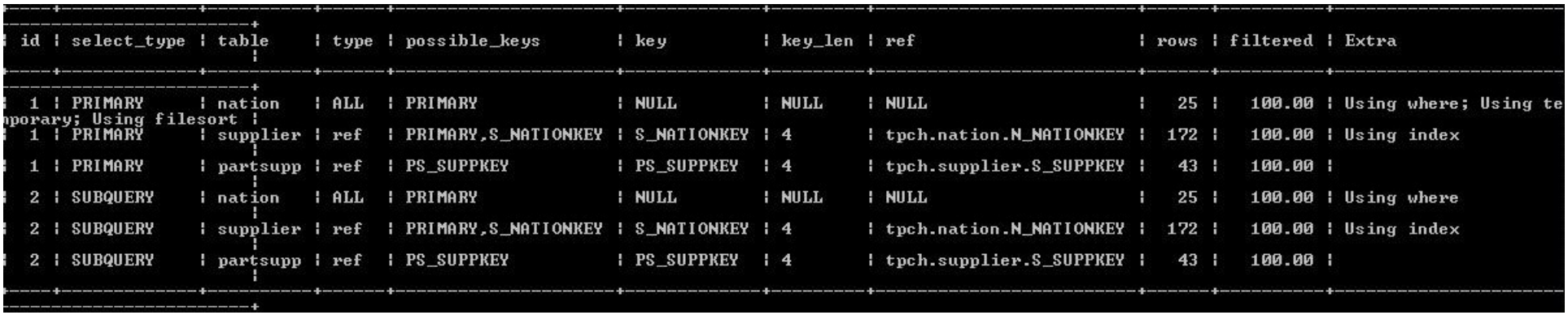
Subsequently, we tried the following: in high server settings, we forced the use of Orderdate index, thus the optimizer used the second plan instead of the first. This time the query ran in 34 seconds! This is an interesting case, as it turns out that the use of as many indexes as the predicates allow does not always lead to better performance. Evaluating the predicate within a scan can be quicker than using two indexes and joining the results; especially in this case when we also needed to sort

the results in the end. Therefore, when SQL Server chooses not to use an available index, there might be a good reason for this decision and we should make sure that manually forcing the use of an index does indeed enhance performance.



id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	nation	ALL	PRIMARY	NULL	NULL	NULL	25	100.00	Using where; Using temporary; Using filesort
1	PRIMARY	supplier	ref	PRIMARY,S_NATIONKEY	S_NATIONKEY	4	tpch.nation.N_NATIONKEY	172	100.00	Using index
1	PRIMARY	partsupp	ref	PS_SUPPKEY	PS_SUPPKEY	4	tpch.supplier.S_SUPPKEY	43	100.00	
2	SUBQUERY	supplier	ALL	PRIMARY,S_NATIONKEY	S_NATIONKEY	NULL	NULL	10033	100.00	
2	SUBQUERY	nation	eq_ref	PRIMARY	PRIMARY	4	tpch.supplier.S_NATIONKEY	1	100.00	Using where
2	SUBQUERY	partsupp	ref	PS_SUPPKEY	PS_SUPPKEY	4	tpch.supplier.S_SUPPKEY	43	100.00	

Figure 8.8: Execution plan for query 11 in MySQL with optimizer_prune_level=0.



id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	nation	ALL	PRIMARY	NULL	NULL	NULL	25	100.00	Using where; Using temporary; Using filesort
1	PRIMARY	supplier	ref	PRIMARY,S_NATIONKEY	S_NATIONKEY	4	tpch.nation.N_NATIONKEY	172	100.00	Using index
1	PRIMARY	partsupp	ref	PS_SUPPKEY	PS_SUPPKEY	4	tpch.supplier.S_SUPPKEY	43	100.00	
2	SUBQUERY	nation	ALL	PRIMARY	NULL	NULL	NULL	25	100.00	Using where
2	SUBQUERY	supplier	ref	PRIMARY,S_NATIONKEY	S_NATIONKEY	4	tpch.nation.N_NATIONKEY	172	100.00	Using index
2	SUBQUERY	partsupp	ref	PS_SUPPKEY	PS_SUPPKEY	4	tpch.supplier.S_SUPPKEY	43	100.00	

Figure 8.9: Execution plan for query 11 in MySQL with optimizer_prune_level=1.

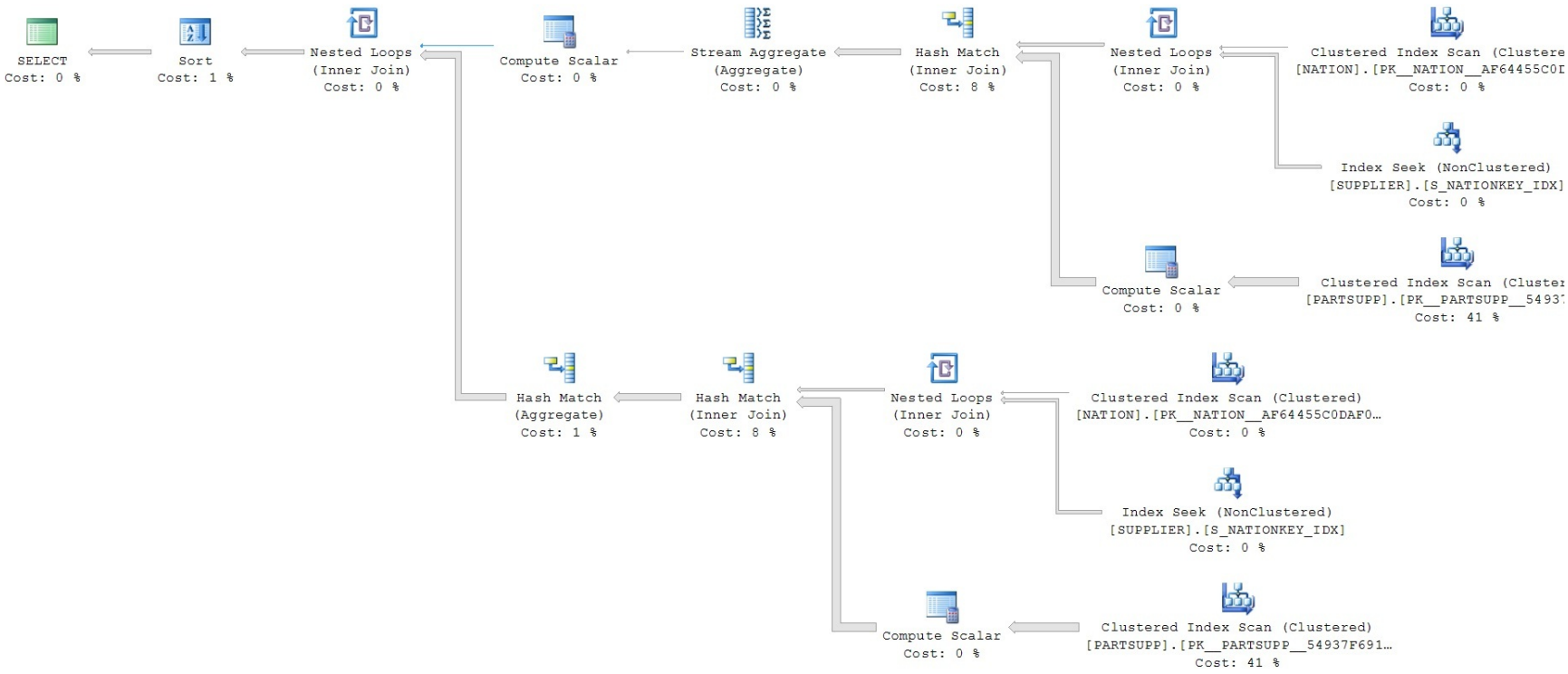


Figure 8.10: Execution plan for query 11 in SQL Server.

```
select
        ps_partkey,
        sum(ps_supplycost * ps_availqty) as value
from
        partsupp,
        supplier,
        nation
where
        ps_suppkey = s_suppkey
        and s_nationkey = n_nationkey
        and n_name = 'UNITED KINGDOM'
group by
        ps_partkey having
                sum(ps_supplycost * ps_availqty) > (
                        select
                                sum(ps_supplycost * ps_availqty) * 0.0001000000
                        from
                                partsupp,
                                supplier use index(primary),
                                nation
                        where
                                ps_suppkey = s_suppkey
                                and s_nationkey = n_nationkey
                                and n_name = 'UNITED KINGDOM'
                )
order by
        value desc;
```

Figure 8.11: Query 11 text.

8.2.2 Experiments with MySQL

In this experiment, we attempted changing the value of the parameter optimizer_prune_level in MySQL. This variable controls the heuristics applied during query optimization to prune less-promising partial plans from the optimizer search space. In other words, it tells the optimizer to perform an 'educated guess' and skip certain plans based on estimates of the number of rows accessed for each table. By default, this option is on (the value of the parameter is 1).

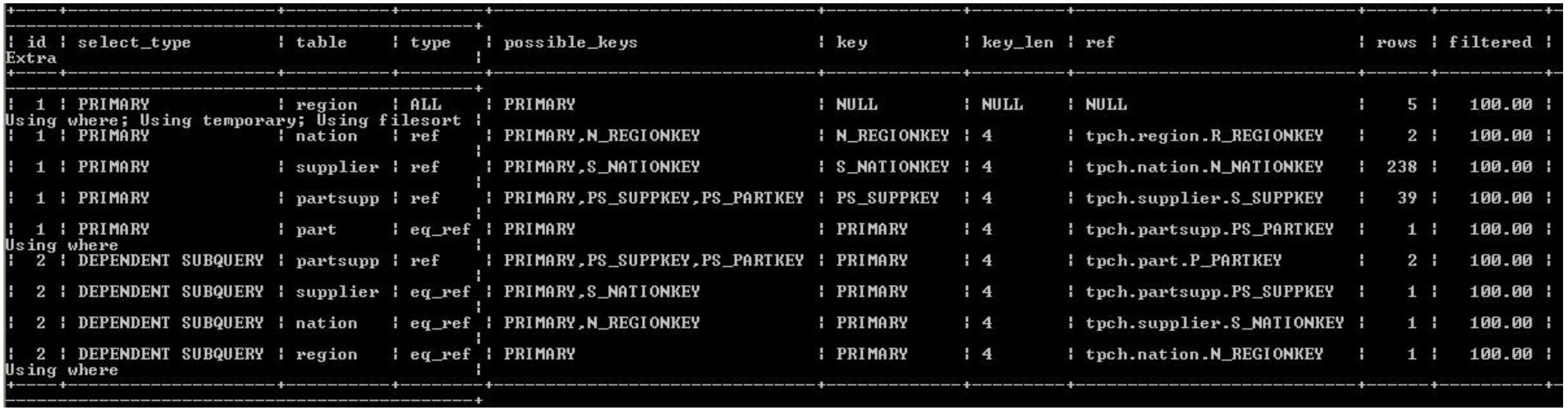
In order to test this function, we used the scan-bound query 11. The generated query plans for MySQL are shown in Figures 8.8 and 8.9, the query plan for SQL Server in Figure 8.10, and the query text in Figure 8.11.

We can observe that changing the value of optimizer_prune_level influences the join order of the tables Supplier and Nation within the subquery. In the first case, MySQL needs to examine $10033+1 = 10034$ rows, while in the second $25+172 = 197$ rows. The fact that in the second case the optimizer took the time to perform a more exhaustive search, rather than simply follow MySQL's practice of putting the largest table first, paid off: in the first case, the query ran in 42 seconds, while in the second in 36 seconds. Thus, DSS queries could benefit from setting the optimizer_prune_level parameter to 0. Also, if we had eliminated this query and realized that it would benefit from a change in join order, it would have made sense to use a join hint, such as straight_join.

Meanwhile, SQL Server's optimizer came up with essentially the same query plan as the second case in MySQL. However, it executed the query in 34 seconds, which may indicate that the optimization procedure itself in SQL Server might be not only reliable but also quick.

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	region	ALL	PRIMARY	NULL	NULL	NULL	5	100.00	Using where; Using temporary; Using filesort
1	PRIMARY	nation	ref	PRIMARY,N_REGIONKEY	N_REGIONKEY	4	tpch.region.R_REGIONKEY	2	100.00	
1	PRIMARY	supplier	ref	PRIMARY,S_NATIONKEY	S_NATIONKEY	4	tpch.nation.N_NATIONKEY	238	100.00	
1	PRIMARY	partsupp	ref	PRIMARY,PS_SUPPKEY,PS_PARTKEY	PS_SUPPKEY	4	tpch.supplier.S_SUPPKEY	39	100.00	
1	PRIMARY	part	eq_ref	PRIMARY	PRIMARY	4	tpch.partsupp.PS_PARTKEY	1	100.00	Using where
2	DEPENDENT SUBQUERY	region	ALL	PRIMARY	NULL	NULL	NULL	5	100.00	Using where
2	DEPENDENT SUBQUERY	nation	ref	PRIMARY,N_REGIONKEY	N_REGIONKEY	4	tpch.region.R_REGIONKEY	2	100.00	Using index
2	DEPENDENT SUBQUERY	partsupp	ref	PRIMARY,PS_SUPPKEY,PS_PARTKEY	PRIMARY	4	tpch.part.P_PARTKEY	2	100.00	
2	DEPENDENT SUBQUERY	supplier	eq_ref	PRIMARY,S_NATIONKEY	PRIMARY	4	tpch.partsupp.PS_SUPPKEY	1	100.00	Using where

Figure 8.12: Execution plan for query 2 in MySQL with optimizer_search_depth=1.



id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	region	ALL	PRIMARY	NULL	NULL	NULL	5	100.00	Using where; Using temporary; Using filesort
1	PRIMARY	nation	ref	PRIMARY,N_REGIONKEY	N_REGIONKEY	4	tpch.region.R_REGIONKEY	2	100.00	
1	PRIMARY	supplier	ref	PRIMARY,S_NATIONKEY	S_NATIONKEY	4	tpch.nation.N_NATIONKEY	238	100.00	
1	PRIMARY	partsupp	ref	PRIMARY,PS_SUPPKEY,PS_PARTKEY	PS_SUPPKEY	4	tpch.supplier.S_SUPPKEY	39	100.00	
1	PRIMARY	part	eq_ref	PRIMARY	PRIMARY	4	tpch.partsupp.PS_PARTKEY	1	100.00	Using where
2	DEPENDENT SUBQUERY	partsupp	ref	PRIMARY,PS_SUPPKEY,PS_PARTKEY	PRIMARY	4	tpch.part.P_PARTKEY	2	100.00	
2	DEPENDENT SUBQUERY	supplier	eq_ref	PRIMARY,S_NATIONKEY	PRIMARY	4	tpch.partsupp.PS_SUPPKEY	1	100.00	
2	DEPENDENT SUBQUERY	nation	eq_ref	PRIMARY,N_REGIONKEY	PRIMARY	4	tpch.supplier.S_NATIONKEY	1	100.00	
2	DEPENDENT SUBQUERY	region	eq_ref	PRIMARY	PRIMARY	4	tpch.nation.N_REGIONKEY	1	100.00	Using where

Figure 8.13: Execution plan for query 2 in MySQL with optimizer_search_depth=62.

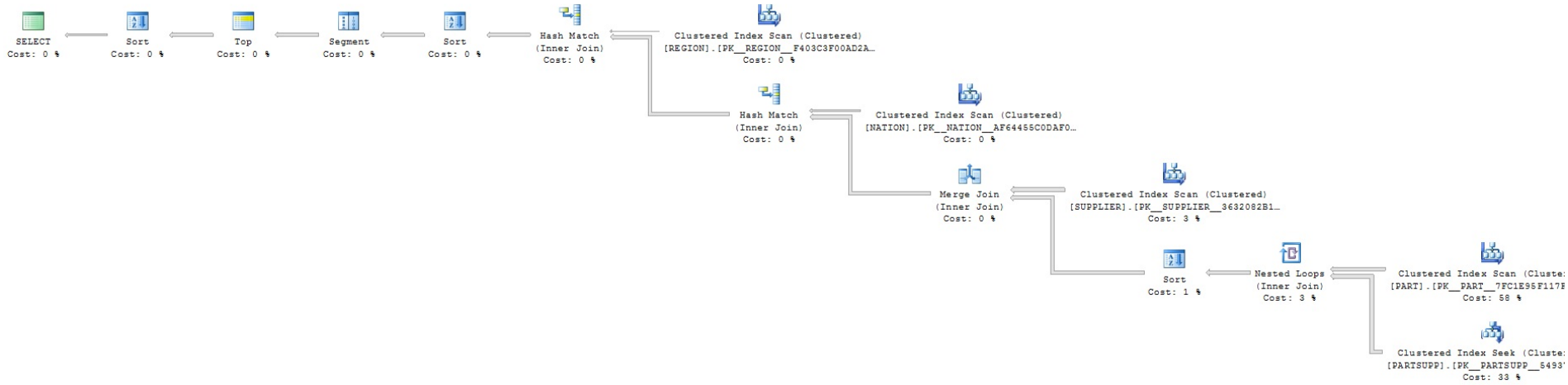


Figure 8.14: Execution plan for query 2 in SQL Server.

```
select
        s_acctbal,
        s_name,
        n_name,
        p_partkey,
        p_mfgr,
        s_address,
        s_phone,
        s_comment
from
        part,
        supplier,
        partsupp,
        nation,
        region
where
        p_partkey = ps_partkey
        and s_suppkey = ps_suppkey
        and p_size = 35
        and p_type like '%STEEL'
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'EUROPE'
        and ps_supplycost = (
                select
                        min(ps_supplycost)
                from
                        partsupp,
                        supplier,
                        nation,
                        region
                where
                        p_partkey = ps_partkey
                        and s_suppkey = ps_suppkey
                        and s_nationkey = n_nationkey
                        and n_regionkey = r_regionkey
                        and r_name = 'EUROPE'
        )
order by
        s_acctbal desc,
        n_name,
        s_name,
        p_partkey
limit 100;
```

Figure 8.15: Query 2 text.

In this experiment, we changed the value of the variable optimizer_search_depth, which controls the maximum depth of search performed by the query optimizer. In other words, this variable tells the optimizer how far into the 'future' of each incomplete plan it should look, to evaluate whether it should be expanded further. Values larger than the number of relations in a query result in better query plans, but take longer to generate an execution plan for a query. Values smaller than the number of relations return an execution plan quicker, but it may be far from being optimal. By default, this variable is set to 62, thus dictating full depth search.

We chose query 2, a join-bound query involving six tables. The execution plan for optimizer_search_depth=1 appears in Figure 8.12, the plan for optimizer_search_depth=62 in Figure 8.13 and the query text in Figure 8.15. Setting the variable equal to 1 in the first experiment is equivalent to setting the lowest possible search depth. The second execution plan that required an in-depth exploration of the query tree search space executed in 342 seconds, whilst the first, naive plan executed in 793 seconds. Therefore, setting this variable at a high value when executing a highly complex query can result in greater performance overall, thanks to a more efficient query plan, even though the optimization process takes longer.

What is more, this query was executed in SQL Server extremely quickly: in only 8 seconds. The execution plan is shown in Figure 8.14. The reason why SQL Server performed so much better in this case, is because it only accessed the tables once, satisfying the predicates for both the query and the subquery. SQL Server is able to spot and take advantage of such opportunities, as it has a special 'normalization'

process during its query parsing phase. The conclusion is that even the most exhaustive search for the optimal join order and index selection in MySQL sometimes simply cannot compete with the so-much-less query-plan-restrictive philosophy of SQL Server optimizer.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

First of all, after examining the TPC-H benchmark in detail and having ran the tests, we can conclude that this is a test that can be performed at home, without a team of experts. Even though our results cannot be compared to the official results due to the low scale factor we had to use, we could still observe differences between different systems and configurations. Therefore, this test is scalable and easy-to-use.

The TPC-H test gave us the motivation to look a little deeper into the factors that influence the performance of DSS queries. These factors include some tuning options, as well as parameters that influence query optimization. We experimented with SQL Server 2008 and MySQL 5.1 while varying such tuning options and parameters, arriving to some interesting conclusions.

The most influential tuning option is the memory size, namely the buffer pool and the sort buffer sizes. Increasing the memory allows more data to be available in the cache, resulting in less need for I/Os, and creates more space for executing large sorts. Essentially, a larger memory size allows us to host more data pages in the cache. Going a little further than that, the fact that DSS workloads tend to contain large scans with high selectivity indicates that it is highly probable that each page contains a lot of relevant data. This means that we are not only interested in the number of pages in the cache, but also in the amount of data they host. Hence, the page size and the fill factor can also play an important role in the performance of DSS queries; and in that order, since increasing the page size is equivalent to a more dramatic increase of data per page.Incidentally, increasing the fill factor does not usually lead to disorganization, as DSS workloads contain few, if any, updates. Both database systems offer options for changing the memory size. However, MySQL offers the page size option, which prevails against SQL Server's fill factor option. Finally, we can use these tuning parameters to enhance performance up to

a point: once all the data has been loaded into the cache and there is enough space for sorts, their influence ceases.

Since the two systems do not have identical configuration options, we cannot tune them fairly. Nevertheless, for similar configurations, MySQL is consistently slower than SQL Server. 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.

Next, we examined the query optimizers, observing how the partially-heuristic philosophy of MySQL generates good enough plans almost always. However, in the case of extremely complex queries like those of TPC-H, SQL Server most of the times comes up with a better execution plan. A reason for this might be the fact that, when calculating a plan cost, only SQL Server query optimizer takes into account the cache size and the fill factor; another reason is that SQL Server samples more pages during statistics creation, therefore it has more accurate information for costing. On top of that, the MySQL optimizer features a parameter called optimizer_prune_level that commands it to ignore some query plans for the sake of quick query optimization. This technique would be useful for OLTP workloads, but it is dangerous for DSS ones. Similar dangers lurk in setting the optimizer_search_depth too low, thus limiting how far into the 'future' of each incomplete plan the optimizer should look. Meanwhile, SQL Server optimizes non-trivial queries in phases, thus automatically deciding to perform a more exhaustive search if the query is highly complex. Finally, we can conclude that DSS queries may be expensive to optimize, yet a good query plan can be highly beneficial for performance. One should monitor the optimizer behaviour and provide query plan hints if required.

9.2 Future Work

Firtsly, 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 at home. In addition, with a larger database we would be able to observe finer differences between different configurations and reach more conclusions.

Secondly, in chapter 6 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. Furthermore, there could 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.

Bibliography

[1] TPC Benchmark^{TM} H (Decision Support) Standard Specification, Revision 2.3.0

[2] Gray, J.,"Database and Transaction Processing Performance Handbook"(2nd Edition), Morgan Kaufmann 1993

[3] Shao, M., Ailamaki, A., Falsafi, B., "DBmbench: Fast and Accurate Database Workload Representation on Modern Microarchitectur", Conference of the Centre for Advanced Studies on Collaborative Research 2005

[4] Chaudhri, A., Rashid, A., Zicari, R., "XML Data Management: Native XML and XML-Enabled Database Systems", Pearson Education 2003

[5] Anon et. al., "A Measure of Transaction Processing Power", Datamation, V.31.7, 1985

[6] Hellerstein, J., Stonebraker, M., Hamilton, J., "Architecture of a Database System", Foundations and Trends in Databases, Vol.1, No.2, 2007

[7] Chaudhuri, S., "An Overview of Query Optimization in Relational Systems", Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, p.34-43, June 01-04, 1998, Seattle

[8] Elnaffar, S., Martin, P., Schiefer, B., Lightstone, S., "Is it DSS or OLTP: automatically identifying DBMS workloads", Journal of Intelligent Information Systems, v.30 n.3, p.249-271, 2008

[9] Elnaffar, S., "A methodology for auto-recognizing DBMS workloads", Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research, p.2, September 30-October 03, 2002, Toronto

[10] Elnaffar, S., Martin, P., Horman, R., "Automatically classifying database workloads", Proceedings of the eleventh international conference on Information and knowledge management, November 04-09, 2002, McLean

[11] Chaudhuri , S., Narasayya, V., "Self-tuning database systems: a decade of progress", Proceedings of the 33rd international conference on Very large data bases, September 23-27, 2007, Vienna, Austria

[12] Wiese1, D., Rabinovitch, G., Reichert, M., Arenswald, S., "Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning", Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, session: databases, article 3, 2008, Ontario

[13] "DB2 Universal Database Version 7 Administration Guide: Performance", IBM Corporation, 2000

[14] "Oracle9i Database Performance Guide and Reference", Release 1(9.0.1), Part A87503-02, Oracle Corp., 2001

[15] Paulsell, K., "Sybase Adaptive Server Enterprise Performance and Tuning Guide", Sybase Inc., Adaptive Server Enterprise Version 12, Document ID: 32614-01-1200-01-01/32615-01-1200-01, 1999

[16] Packer, A., "Configuring and Tuning Databases on the Solaris Platform", Sun Microsystems Press, ISBN 0-13-083417-3, 2002

[17] Chokshi, D., "Performance Comparison Between ASE 15.0 and MySQL 5.0 White Paper", Server Performance and Engineering Development Group, Sybase Inc., 2006

[18] Zhou, J., "Database Performance Analysis and Tuning: A Comparative Study of TPC-H Benchmark on Oracle and DB2", Concordia University, 2003, Montreal

[19] Castanhede, T., Dill, S., Padoin, E., "Performance Evaluation of MySQL under different file systems" (in Portuguese), IV Sulcomp, Santa Catarina, 2008

[20] Mullins, C., "Database Administration: The Complete Guide to Practices and Procedures", Addison-Wesley, 2002

[21] Selinger, P., Astraham, D., Lories, R., Price, T., "Access Path Selection in a Relational Database Management System", Proceedings of the ACM SIGMOD International Conference on the Management of Data, p.23-24, Aberdeen, 1979

[22] Stonebraker, M., Wong, E., Kreps, P., "The Design and Implementation of INGRES", ACM Transactions on Database Systems 1(3), p.189-222, 1976

[23] McKenna, W., "Volcano Query Optimizer Generator Manual", University of Colorado, 1992

[24] Ioannidis, Y., Ng, R., Shim, K., Sellis, T., "Parametric Query Optimization", The VLDB Journal • The International Journal on Very Large Data Bases, v.6 n.2, p.132-151, 1997

[25] Bell, C., "Expert MySQL", Apress, 2007

[26] Delaney, K. et. al., "Microsoft SQL Server 2008 Internals", Microsoft Press, 2009

[27] Graefe, G., "The Cascades Framework for Query Optimization", IEEE Data Eng. Bull. 18(3), p.19-29, 1995

[28] Harizopoulos, S., Shkapenyuk , V., Ailamaki, A., "QPipe: a simultaneously pipelined relational query engine", Proceedings of the 2005 ACM SIGMOD international conference on Management of data,p.14-16, Baltimore, 2005

[29] Idreos, S., Kersten, M., Manegold, S., "Self-organizing tuple reconstruction in column-stores", Proceedings of the 35th SIGMOD international conference on Management of data, Providence, 2009

[30] Acar, A., Motro, A., "Efficient discovery of join plans in schemaless data", Proceedings of the 2009 International Database Engineering & Applications Symposium, p.1-11, Cetraro, 2009

[31] Wiese, D., Rabinovitch, G., Reichert, M., Arenswald, S., "Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning", Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, article 3, Ontario, 2008

[32] Ailamaki, A., DeWitt, D., Hill, M., "Data page layouts for relational databases on deep memory hierarchies", The VLDB Journal • The International Journal on Very Large Data Bases, v.11 n.3, p.198-215, 2002

[33] Somogyi, S., Wenisch, T., Ailamaki , A., Falsafi, B., "Spatio-Temporal Memory Streaming", Proceedings of the 36th annual international symposium on Computer Architecture, p.69-80, Austin, 2009

[34] Johnson, R., Harizopoulos , S., Hardavellas, N., Sabirli, K., Pandis, I., Ailamaki, A., Mancheril, N., Falsafi, B., "To share or not to share?", Proceedings of the 33rd international conference on Very large data bases, Vienna, 2007

[35] Guehis, S., Goasdoue-Thion, V., Rigaux, P., "Speeding-up data-driven applications with program summaries", Proceedings of the 2009 International Database Engineering & Applications Symposium, p.66-76, Cetraro, 2009

[36] Kitsuregawa, M., Goda, K., Hoshino, T., "Storage fusion", Proceedings of the 2nd international conference on Ubiquitous information management and communication, p.270-277, Suwon, 2008

[37] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S., Wenisch, T., "Disaggregated memory for expansion and sharing in blade servers", Proceedings of the 36th annual international symposium on Computer architecture, p.267-278, Austin, 2009

[38] Trigoni, A., Moody, K., "Using Association Rules to Add or Eliminate Query Constraints Automatically", 13th International Conference on Scientific and Statistical Database Management, 2001

[39] Aberer, K., Fischer, G., "Semantic Query Optimization for Methods in Object-Oriented Database Systems", Gesellschaft fuer Mathematik und Datenverarbeitung (GMD), Darmstadt, 1994

[40] Genet B., Dobbie, G., "Is semantic optimisation worthwhile?", In Proceedingsof the 21st Australasian Computer Science Conference, p. 245•256, Perth, 1998

[41] Mannino, M., "Database Design, Application Development, and Administration", McGraw-Hill, 2004

[42] King, J., "QUIST: a system for semantic query optimization in relational databases", Proceedings of the 7th International Conference on Very Large Databases, p.510-517, Los Angeles, 1981

Appendix A

Source code for Microsoft SQL Server 2008

A.1 Database Build Scripts

CreateTables.sql

```
create table PART
(P_PARTKEY int not null,
P_NAME varchar(55) not null,
P_MFGR char(25) not null,
P_BRAND char(10) not null,
P_TYPE varchar(25) not null,
P_SIZE int not null,
P_CONTAINER char(10) not null,
P_RETAILPRICE float not null,
P_COMMENT varchar(23) not null)

create table SUPPLIER
(S_SUPPKEY int not null,
S_NAME char(25) not null,
S_ADDRESS varchar(40) not null,
S_NATIONKEY int not null,
S_PHONE char(15) not null,
S_ACCTBAL float not null,
S_COMMENT varchar(101) not null)

create table PARTSUPP
(PS_PARTKEY int not null,
PS_SUPPKEY int not null,
PS_AVAILQTY int not null,
PS_SUPPLYCOST float not null,
PS_COMMENT varchar(199) not null)

create table CUSTOMER
(C_CUSTKEY int not null,
C_NAME varchar(25) not null,
C_ADDRESS varchar(40) not null,
C_NATIONKEY int not null,
C_PHONE char(15) not null,
C_ACCTBAL float not null,
C_MKTSEGMENT char(10) not null,
C_COMMENT varchar(117) not null)

create table ORDERS
(O_ORDERKEY bigint not null,
O_CUSTKEY int not null,
O_ORDERSTATUS char(1) not null,
O_TOTALPRICE float not null,
O_ORDERDATE date not null,
O_ORDERPRIORITY char(15) not null,
O_CLERK char(15) not null,
O_SHIPPRIORITY int not null,
O_COMMENT varchar(79) not null)

create table LINEITEM
(L_ORDERKEY bigint not null,
L_PARTKEY int not null,
L_SUPPKEY int not null,
L_LINENUMBER int not null,
L_QUANTITY float not null,
L_EXTENDEDPRICE float not null,
L_DISCOUNT float not null,
L_TAX float not null,
L_RETURNFLAG char(1) not null,
L_LINESTATUS char(1) not null,
L_SHIPDATE date not null,
L_COMMITDATE date not null,
L_RECEIPTDATE date not null,
L_SHIPINSTRUCT char(25) not null,
L_SHIPMODE char(10) not null,
L_COMMENT varchar(44) not null)

create table NATION
(N_NATIONKEY int not null,
N_NAME char(25) not null,
N_REGIONKEY int not null,
N_COMMENT varchar(152) not null)

create table REGION
(R_REGIONKEY int not null,
R_NAME char(25) not null,
R_COMMENT varchar(152) not null)

create table TIMES
(QUERY char(25),
START datetime)

create table NEWORDERS
(O_ORDERKEY bigint not null,
O_CUSTKEY int not null,
O_ORDERSTATUS char(1) not null,
O_TOTALPRICE float not null,
O_ORDERDATE datetime not null,
O_ORDERPRIORITY char(15) not null,
O_CLERK char(15) not null,
O_SHIPPRIORITY int not null,
O_COMMENT varchar(79) not null)

create table NEWLINEITEM
(L_ORDERKEY bigint not null,
L_PARTKEY int not null,
L_SUPPKEY int not null,
L_LINENUMBER int not null,
L_QUANTITY float not null,
L_EXTENDEDPRICE float not null,
L_DISCOUNT float not null,
L_TAX float not null,
L_RETURNFLAG char(1) not null,
L_LINESTATUS char(1) not null,
L_SHIPDATE datetime not null,
L_COMMITDATE datetime not null,
L_RECEIPTDATE datetime not null,
L_SHIPINSTRUCT char(25) not null,
L_SHIPMODE char(10 ) not null,
L_COMMENT varchar(44) not null)

create table OLDORDERS
(O_ORDERKEY int not null)
```

Constraints.sql

```
-- For table REGION
```

```
ALTER TABLE TPCH.dbo.REGION
ADD PRIMARY KEY (R_REGIONKEY);

-- For table NATION
ALTER TABLE TPCH.dbo.NATION
ADD PRIMARY KEY (N_NATIONKEY);

ALTER TABLE TPCH.dbo.NATION
ADD FOREIGN KEY (N_REGIONKEY)
references TPCH.dbo.REGION;

-- For table PART
ALTER TABLE TPCH.dbo.PART
ADD PRIMARY KEY (P_PARTKEY);

-- For table SUPPLIER
ALTER TABLE TPCH.dbo.SUPPLIER
ADD PRIMARY KEY (S_SUPPKEY);

ALTER TABLE TPCH.dbo.SUPPLIER
ADD FOREIGN KEY (S_NATIONKEY)
references TPCH.dbo.NATION;

-- For table PARTSUPP
ALTER TABLE TPCH.dbo.PARTSUPP
ADD PRIMARY KEY (PS_PARTKEY,PS_SUPPKEY);

-- For table CUSTOMER
ALTER TABLE TPCH.dbo.CUSTOMER
ADD PRIMARY KEY (C_CUSTKEY);

ALTER TABLE TPCH.dbo.CUSTOMER
ADD FOREIGN KEY (C_NATIONKEY)
references TPCH.dbo.NATION;

-- For table LINEITEM
ALTER TABLE TPCH.dbo.LINEITEM
ADD PRIMARY KEY (L_ORDERKEY,
L_LINENUMBER);

-- For table ORDERS
ALTER TABLE TPCH.dbo.ORDERS
ADD PRIMARY KEY (O_ORDERKEY);

-- For table PARTSUPP
ALTER TABLE TPCH.dbo.PARTSUPP
ADD FOREIGN KEY (PS_SUPPKEY)
references TPCH.dbo.SUPPLIER;

ALTER TABLE TPCH.dbo.PARTSUPP
ADD FOREIGN KEY (PS_PARTKEY)
references TPCH.dbo.PART;

-- For table ORDERS
ALTER TABLE TPCH.dbo.ORDERS
ADD FOREIGN KEY (O_CUSTKEY)
references TPCH.dbo.CUSTOMER;

-- For table LINEITEM
ALTER TABLE TPCH.dbo.LINEITEM
ADD FOREIGN KEY (L_ORDERKEY)
references TPCH.dbo.ORDERS;

ALTER TABLE TPCH.dbo.LINEITEM
ADD FOREIGN KEY (L_PARTKEY,L_SUPPKEY)
references TPCH.dbo.PARTSUPP;
```

Indexes.sql

```
CREATE INDEX S_NATIONKEY_IDX
ON SUPPLIER(S_NATIONKEY);

CREATE INDEX PS_PARTKEY_IDX
ON PARTSUPP(PS_PARTKEY);

CREATE INDEX PS_SUPPKEY_IDX
ON PARTSUPP(PS_SUPPKEY);

CREATE INDEX C_NATIONKEY_IDX
ON CUSTOMER(C_NATIONKEY);

CREATE INDEX O_CUSTKEY_IDX
ON ORDERS(O_CUSTKEY);

CREATE INDEX L_ORDERKEY_IDX
ON LINEITEM(L_ORDERKEY);

CREATE INDEX L_PARTKEY_IDX
ON LINEITEM(L_PARTKEY);

CREATE INDEX L_SUPPKEY_IDX
ON LINEITEM(L_SUPPKEY);

CREATE INDEX N_REGIONKEY_IDX
ON NATION(N_REGIONKEY);

CREATE INDEX O_ORDERDATE_CLUIDX
ON ORDERS(O_ORDERDATE);

CREATE INDEX L_SHIPDATE_CLUIDX
ON LINEITEM(L_SHIPDATE);

CREATE INDEX L_COMMITDATE_CLUIDX
ON LINEITEM(L_COMMITDATE);

CREATE INDEX L_RECEIPTDATE_CLUIDX
ON LINEITEM(L_RECEIPTDATE);

CREATE CLUSTERED INDEX NEWORDERS_CLUIDX
ON NEWORDERS (O_ORDERKEY);

CREATE CLUSTERED INDEX NEWLINEITEM_CLUIDX
ON NEWLINEITEM (L_ORDERKEY);

CREATE CLUSTERED INDEX OLDORDERS_CLUIDX
ON OLDORDERS (O_ORDERKEY);
```

A.2 Refresh Function Definitions

CreateRF1.sql

```
CREATE PROCEDURE RF1
@startkey INTEGER
AS
BEGIN

DECLARE @loops INTEGER
DECLARE @orderSql NCHAR(1000)
DECLARE @liSql NCHAR(1000)
DECLARE @success INTEGER

SET @orderSql = N'INSERT INTO ORDERS
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = @startkey'

SET @liSql = N'INSERT INTO LINEITEM
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
WHERE L_ORDERKEY = O_ORDERKEY
AND O_ORDERKEY = @startkey'

SET @loops = 124

WHILE @loops > 0
BEGIN
INSERT_TRANS:
SET @success = 1
BEGIN TRANSACTION;
BEGIN TRY;

EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @orderSql,
```

```
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;

END TRY

BEGIN CATCH
SET @success = 0
IF (error_number() = 1205) -- deadlock
PRINT 'Insert deadlock - restarting RF1'
ELSE
BEGIN -- not a deadlock
PRINT 'Error - Not a deadlock'
PRINT ERROR_NUMBER()
PRINT ERROR_SEVERITY()
PRINT ERROR_MESSAGE()
PRINT ERROR_STATE()
PRINT XACT_STATE()
END
IF (@@trancount>0)
ROLLBACK TRANSACTION;
END CATCH
IF (@success = 0) -- deadlock
GOTO INSERT_TRANS
COMMIT TRANSACTION;

SET @startkey = (@startkey + 25)
SET @loops = (@loops - 1)
END

END
GO
```

CreateRF2.sql

```
CREATE PROCEDURE RF2
@startkey INTEGER
AS
BEGIN

DECLARE @loops INTEGER
DECLARE @orderSql NCHAR(1000)
DECLARE @liSql NCHAR(1000)
DECLARE @success INTEGER

SET @orderSql = N'DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = @startkey)'

SET @liSql = N'DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = @startkey)'

SET @loops = 124

WHILE @loops > 0
BEGIN
INSERT_TRANS:
SET @success = 1
BEGIN TRANSACTION;
BEGIN TRY;

EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;

SET @startkey = (@startkey + 1)

EXEC sp_executesql @liSql,
N'@startkey INTEGER', @startkey;
EXEC sp_executesql @orderSql,
N'@startkey INTEGER', @startkey;

END TRY

BEGIN CATCH
SET @success = 0
IF (error_number() = 1205) -- deadlock
PRINT 'Insert deadlock - restarting RF1'
ELSE
BEGIN -- not a deadlock
PRINT 'Error - Not a deadlock'
PRINT ERROR_NUMBER()
```

```
PRINT ERROR_SEVERITY()
PRINT ERROR_MESSAGE()
PRINT ERROR_STATE()
PRINT XACT_STATE()
END
IF (@@trancount>0)
ROLLBACK TRANSACTION;
END CATCH
IF (@success = 0) -- deadlock
GOTO INSERT_TRANS
COMMIT TRANSACTION;
SET @startkey = (@startkey + 25)
SET @loops = (@loops - 1)
END

END
GO
```

A.3 Query Streams

Stream00.sql

```
-- using 1027173119 as a seed to the RNG

create view revenue0
(supplier_no, total_revenue) as
select
l_suppkey,
sum(l_extendedprice * (1 - l_discount))
from
lineitem
where
l_shipdate >=  '1996-04-01'
and l_shipdate < dateadd(month,
+3, '1996-04-01')
group by
l_suppkey;
go

insert into TIMES values
('Str00 start',getdate());

insert into TIMES values
('Q14 in Str00 start',getdate());
select
100.00 * sum(case
when p_type like 'PROMO%'
then l_extendedprice * (1 - l_discount)
else 0
end) / sum(l_extendedprice * (1 - l_discount))
as promo_revenue
from
lineitem,
part
where
l_partkey = p_partkey
and l_shipdate >=  '1995-08-01'
and l_shipdate < dateadd(month,
+1, '1995-08-01');
go
insert into TIMES values
('Q14 in Str00 end',getdate());

set rowcount 100
go

insert into TIMES values
('Q2 in Str00 start',getdate());
select
s_acctbal,
s_name,
n_name,
p_partkey,
p_mfgr,
s_address,
s_phone,
s_comment
from
part,
supplier,
partsupp,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and p_size = 27
and p_type like '%TIN'
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'MIDDLE EAST'
and ps_supplycost = (
select
min(ps_supplycost)
from
partsupp,
supplier,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'MIDDLE EAST'
)
order by
s_acctbal desc,
n_name,
s_name,
p_partkey;
go
insert into TIMES values
('Q2 in Str00 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q9 in Str00 start',getdate());
select
nation,
o_year,
sum(amount) as sum_profit
from
(
select
n_name as nation,
datepart(year,o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
- ps_supplycost * l_quantity as amount
from
part,
supplier,
lineitem,
partsupp,
orders,
nation
where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like '%dark%'
) as profit
group by
nation,
o_year
order by
nation,
o_year desc;
go
insert into TIMES values
('Q9 in Str00 end',getdate());

insert into TIMES values
('Q20 in Str00 start',getdate());
select
s_name,
s_address
from
supplier,
nation
where
s_suppkey in (
select
ps_suppkey
from
```

```
partsupp
where
ps_partkey in (
select
p_partkey
from
part
where
p_name like 'maroon%'
)
and ps_availqty > (
select
0.5 * sum(l_quantity)
from
lineitem
where
l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and l_shipdate >=  '1997-01-01'
and l_shipdate < dateadd(year,
+1, '1997-01-01')
)
)
and s_nationkey = n_nationkey
and n_name = 'FRANCE'
order by
s_name;
go
insert into TIMES values
('Q20 in Str00 end',getdate());

insert into TIMES values
('Q6 in Str00 start',getdate());
select
sum(l_extendedprice * l_discount) as revenue
from
lineitem
where
l_shipdate >=  '1993-01-01'
and l_shipdate < dateadd(year,
+1, '1993-01-01')
and l_discount between 0.06 - 0.01
and 0.06 + 0.01
and l_quantity < 25;
go
insert into TIMES values
('Q6 in Str00 end',getdate());

insert into TIMES values
('Q17 in Str00 start',getdate());
select
sum(l_extendedprice) / 7.0 as avg_yearly
from
lineitem,
part
where
p_partkey = l_partkey
and p_brand = 'Brand#52'
and p_container = 'MED DRUM'
and l_quantity < (
select
0.2 * avg(l_quantity)
from
lineitem
where
l_partkey = p_partkey
);
go
insert into TIMES values
('Q17 in Str00 end',getdate());

set rowcount 100
go

insert into TIMES values
('Q18 in Str00 start',getdate());
select
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice,
sum(l_quantity)
from
customer,
orders,
lineitem
where
o_orderkey in (
select
l_orderkey
from
lineitem
group by
l_orderkey having
sum(l_quantity) > 315
)
and c_custkey = o_custkey
and o_orderkey = l_orderkey
group by
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice
order by
o_totalprice desc,
o_orderdate;
go
insert into TIMES values
('Q18 in Str00 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q8 in Str00 start',getdate());
select
o_year,
sum(case
when nation = 'RUSSIA' then volume
else 0
end) / sum(volume) as mkt_share
from
(
select
datepart(year,o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
as volume,
n2.n_name as nation
from
part,
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2,
region
where
p_partkey = l_partkey
and s_suppkey = l_suppkey
and l_orderkey = o_orderkey
and o_custkey = c_custkey
and c_nationkey = n1.n_nationkey
and n1.n_regionkey = r_regionkey
and r_name = 'EUROPE'
and s_nationkey = n2.n_nationkey
and o_orderdate between  '1995-01-01'
and  '1996-12-31'
and p_type = 'SMALL BURNISHED COPPER'
) as all_nations
group by
o_year
order by
o_year;
go
insert into TIMES values
('Q8 in Str00 end',getdate());

set rowcount 100
go

insert into TIMES values
('Q21 in Str00 start',getdate());
select
s_name,
count(*) as numwait
from
supplier,
lineitem l1,
orders,
nation
where
```

```
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (
select
*
from
lineitem l2
where
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (
select
*
from
lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
and n_name = 'UNITED STATES'
group by
s_name
order by
numwait desc,
s_name;
go
insert into TIMES values
('Q21 in Str00 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q13 in Str00 start',getdate());
select
c_count,
count(*) as custdist
from
(
select
c_custkey,
count(o_orderkey)
from
customer left outer join orders on
c_custkey = o_custkey
and o_comment not
like '%unusual%requests%'
group by
c_custkey
) as c_orders (c_custkey, c_count)
group by
c_count
order by
custdist desc,
c_count desc;
go
insert into TIMES values
('Q13 in Str00 end',getdate());

set rowcount 10
go

insert into TIMES values
('Q3 in Str00 start',getdate());
select
l_orderkey,
sum(l_extendedprice * (1 - l_discount))
as revenue,
o_orderdate,
o_shippriority
from
customer,
orders,
lineitem
where
c_mktsegment = 'FURNITURE'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate <  '1995-03-30'
and l_shipdate >  '1995-03-30'
group by
l_orderkey,
o_orderdate,
o_shippriority
order by
revenue desc,
o_orderdate;
go
insert into TIMES values
('Q3 in Str00 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q22 in Str00 start',getdate());
select
cntrycode,
count(*) as numcust,
sum(c_acctbal) as totacctbal
from
(
select
substring(c_phone, 1, 2) as cntrycode,
c_acctbal
from
customer
where
substring(c_phone, 1, 2) in
('16', '29', '33', '34', '26', '22', '31')
and c_acctbal > (
select
avg(c_acctbal)
from
customer
where
c_acctbal > 0.00
and substring(c_phone, 1, 2) in
('16', '29', '33', '34', '26', '22', '31')
)
and not exists (
select
*
from
orders
where
o_custkey = c_custkey
)
) as custsale
group by
cntrycode
order by
cntrycode;
go
insert into TIMES values
('Q22 in Str00 end',getdate());

insert into TIMES values
('Q16 in Str00 start',getdate());
select
p_brand,
p_type,
p_size,
count(distinct ps_suppkey) as supplier_cnt
from
partsupp,
part
where
p_partkey = ps_partkey
and p_brand <> 'Brand#10'
and p_type not like 'PROMO ANODIZED%'
and p_size
in (30, 27, 50, 23, 2, 33, 49, 15)
and ps_suppkey not in (
select
s_suppkey
from
supplier
where
s_comment like '%Customer%Complaints%'
)
group by
p_brand,
p_type,
p_size
order by
supplier_cnt desc,
p_brand,
p_type,
```

```
p_size;
go
insert into TIMES values
('Q16 in Str00 end',getdate());

insert into TIMES values
('Q4 in Str00 start',getdate());
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >=  '1997-09-01'
and o_orderdate < dateadd(month,
+3, '1997-09-01')
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
go
insert into TIMES values
('Q4 in Str00 end',getdate());

insert into TIMES values
('Q11 in Str00 start',getdate());
select
ps_partkey,
sum(ps_supplycost * ps_availqty) as value
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'UNITED KINGDOM'
group by
ps_partkey having
sum(ps_supplycost * ps_availqty) > (
select
sum(ps_supplycost * ps_availqty)
* 0.0001000000
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'UNITED KINGDOM'
)
order by
value desc;
go
insert into TIMES values
('Q11 in Str00 end',getdate());

insert into TIMES values
('Q15 in Str00 start',getdate());
select
s_suppkey,
s_name,
s_address,
s_phone,
total_revenue
from
supplier,
revenue0
where
s_suppkey = supplier_no
and total_revenue = (
select
max(total_revenue)
from
revenue0
)
order by
s_suppkey;
go
drop view revenue0;
go
insert into TIMES values
('Q15 in Str00 end',getdate());

insert into TIMES values
('Q1 in Str00 start',getdate());
select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount))
as sum_disc_price,
sum(l_extendedprice * (1 - l_discount)
* (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= dateadd(day,
-66, '1998-12-01')
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
go
insert into TIMES values
('Q1 in Str00 end',getdate());

set rowcount 20
go

insert into TIMES values
('Q10 in Str00 start',getdate());
select
c_custkey,
c_name,
sum(l_extendedprice * (1 - l_discount))
as revenue,
c_acctbal,
n_name,
c_address,
c_phone,
c_comment
from
customer,
orders,
lineitem,
nation
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate >=  '1993-06-01'
and o_orderdate < dateadd(month,
+3, '1993-06-01')
and l_returnflag = 'R'
and c_nationkey = n_nationkey
group by
c_custkey,
c_name,
c_acctbal,
c_phone,
n_name,
c_address,
c_comment
order by
revenue desc;
go
insert into TIMES values
('Q10 in Str00 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q19 in Str00 start',getdate());
select
sum(l_extendedprice* (1 - l_discount))
as revenue
```

```
from
lineitem,
part
where
(
p_partkey = l_partkey
and p_brand = 'Brand#31'
and p_container
in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
and l_quantity >= 10 and l_quantity <= 10 + 10
and p_size between 1 and 5
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#53'
and p_container
in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
and l_quantity >= 17 and l_quantity <= 17 + 10
and p_size between 1 and 10
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#24'
and p_container
in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
and l_quantity >= 20 and l_quantity <= 20 + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
);
go
insert into TIMES values
('Q19 in Str00 end',getdate());

insert into TIMES values
('Q5 in Str00 start',getdate());
select
n_name,
sum(l_extendedprice * (1 - l_discount))
as revenue
from
customer,
orders,
lineitem,
supplier,
nation,
region
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'AMERICA'
and o_orderdate >=  '1993-01-01'
and o_orderdate < dateadd(year,
+1, '1993-01-01')
group by
n_name
order by
revenue desc;
go
insert into TIMES values
('Q5 in Str00 end',getdate());

insert into TIMES values
('Q7 in Str00 start',getdate());
select
supp_nation,
cust_nation,
l_year,
sum(volume) as revenue
from
(
select
n1.n_name as supp_nation,
n2.n_name as cust_nation,
datepart(year,l_shipdate) as l_year,
l_extendedprice * (1 - l_discount)
as volume
from
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2
where
s_suppkey = l_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and (
(n1.n_name = 'MOROCCO'
and n2.n_name = 'RUSSIA')
or (n1.n_name = 'RUSSIA'
and n2.n_name = 'MOROCCO')
)
and l_shipdate between  '1995-01-01'
and  '1996-12-31'
) as shipping
group by
supp_nation,
cust_nation,
l_year
order by
supp_nation,
cust_nation,
l_year;
go
insert into TIMES values
('Q7 in Str00 end',getdate());

insert into TIMES values
('Q12 in Str00 start',getdate());
select
l_shipmode,
sum(case
when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1
else 0
end) as high_line_count,
sum(case
when o_orderpriority <> '1-URGENT'
and o_orderpriority <> '2-HIGH'
then 1
else 0
end) as low_line_count
from
orders,
lineitem
where
o_orderkey = l_orderkey
and l_shipmode in ('RAIL', 'MAIL')
and l_commitdate < l_receiptdate
and l_shipdate < l_commitdate
and l_receiptdate >=  '1995-01-01'
and l_receiptdate < dateadd(year,
+1, '1995-01-01')
group by
l_shipmode
order by
l_shipmode;
go
insert into TIMES values
('Q12 in Str00 end',getdate());
go
insert into TIMES values
('Str00 end',getdate());
go
```

Stream01.sql

```
-- using 1027173120 as a seed to the RNG

create view revenue1
(supplier_no, total_revenue) as
select
l_suppkey,
sum(l_extendedprice * (1 - l_discount))
from
lineitem
where
```

```
l_shipdate >=  '1994-07-01'
and l_shipdate < dateadd(month,
+3, '1994-07-01')
group by
l_suppkey;
go

insert into TIMES values
('Str01 start',getdate());

set rowcount 100
go

insert into TIMES values
('Q21 in Str01 start',getdate());
select
s_name,
count(*) as numwait
from
supplier,
lineitem l1,
orders,
nation
where
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (
select
*
from
lineitem l2
where
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (
select
*
from
lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
and n_name = 'PERU'
group by
s_name
order by
numwait desc,
s_name;
go
insert into TIMES values
('Q21 in Str01 end',getdate());

set rowcount 10
go

insert into TIMES values
('Q3 in Str01 start',getdate());
select
l_orderkey,
sum(l_extendedprice * (1 - l_discount))
as revenue,
o_orderdate,
o_shippriority
from
customer,
orders,
lineitem
where
c_mktsegment = 'MACHINERY'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate <  '1995-03-16'
and l_shipdate >  '1995-03-16'
group by
l_orderkey,
o_orderdate,
o_shippriority
order by
revenue desc,
o_orderdate;
go
insert into TIMES values
('Q3 in Str01 end',getdate());

set rowcount 100
go

insert into TIMES values
('Q18 in Str01 start',getdate());
select
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice,
sum(l_quantity)
from
customer,
orders,
lineitem
where
o_orderkey in (
select
l_orderkey
from
lineitem
group by
l_orderkey having
sum(l_quantity) > 312
)
and c_custkey = o_custkey
and o_orderkey = l_orderkey
group by
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice
order by
o_totalprice desc,
o_orderdate;
go
insert into TIMES values
('Q18 in Str01 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q5 in Str01 start',getdate());
select
n_name,
sum(l_extendedprice * (1 - l_discount))
as revenue
from
customer,
orders,
lineitem,
supplier,
nation,
region
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'ASIA'
and o_orderdate >=  '1994-01-01'
and o_orderdate < dateadd(year,
+1, '1994-01-01')
group by
n_name
order by
revenue desc;
go
insert into TIMES values
('Q5 in Str01 end',getdate());

insert into TIMES values
('Q11 in Str01 start',getdate());
select
ps_partkey,
sum(ps_supplycost * ps_availqty) as value
from
partsupp,
supplier,
```

```
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'IRAQ'
group by
ps_partkey having
sum(ps_supplycost * ps_availqty) > (
select
sum(ps_supplycost * ps_availqty)
* 0.0001000000
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'IRAQ'
)
order by
value desc;
go
insert into TIMES values
('Q11 in Str01 end',getdate());

insert into TIMES values
('Q7 in Str01 start',getdate());
select
supp_nation,
cust_nation,
l_year,
sum(volume) as revenue
from
(
select
n1.n_name as supp_nation,
n2.n_name as cust_nation,
datepart(year, l_shipdate) as l_year,
l_extendedprice * (1 - l_discount) as volume
from
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2
where
s_suppkey = l_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and (
(n1.n_name = 'GERMANY'
and n2.n_name = 'KENYA')
or (n1.n_name = 'KENYA'
and n2.n_name = 'GERMANY')
)
and l_shipdate between  '1995-01-01'
and  '1996-12-31'
) as shipping
group by
supp_nation,
cust_nation,
l_year
order by
supp_nation,
cust_nation,
l_year;
go
insert into TIMES values
('Q7 in Str01 end',getdate());

insert into TIMES values
('Q6 in Str01 start',getdate());
select
sum(l_extendedprice * l_discount)
as revenue
from
lineitem
where
l_shipdate >=  '1994-01-01'
and l_shipdate < dateadd(year,
+1, '1994-01-01')
and l_discount between 0.04 - 0.01
and 0.04 + 0.01
and l_quantity < 25;
go
insert into TIMES values
('Q6 in Str01 end',getdate());

insert into TIMES values
('Q20 in Str01 start',getdate());
select
s_name,
s_address
from
supplier,
nation
where
s_suppkey in (
select
ps_suppkey
from
partsupp
where
ps_partkey in (
select
p_partkey
from
part
where
p_name like 'tomato%'
)
and ps_availqty > (
select
0.5 * sum(l_quantity)
from
lineitem
where
l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and l_shipdate >=  '1996-01-01'
and l_shipdate < dateadd(year,
+1, '1996-01-01')
)
)
and s_nationkey = n_nationkey
and n_name = 'VIETNAM'
order by
s_name;
go
insert into TIMES values
('Q20 in Str01 end',getdate());

insert into TIMES values
('Q17 in Str01 start',getdate());
select
sum(l_extendedprice) / 7.0 as avg_yearly
from
lineitem,
part
where
p_partkey = l_partkey
and p_brand = 'Brand#51'
and p_container = 'JUMBO BAG'
and l_quantity < (
select
0.2 * avg(l_quantity)
from
lineitem
where
l_partkey = p_partkey
);
go
insert into TIMES values
('Q17 in Str01 end',getdate());

insert into TIMES values
('Q12 in Str01 start',getdate());
select
l_shipmode,
sum(case
when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1
else 0
end) as high_line_count,
sum(case
when o_orderpriority <> '1-URGENT'
and o_orderpriority <> '2-HIGH'
then 1
else 0
```

```
end) as low_line_count
from
orders,
lineitem
where
o_orderkey = l_orderkey
and l_shipmode in ('AIR', 'MAIL')
and l_commitdate < l_receiptdate
and l_shipdate < l_commitdate
and l_receiptdate >=  '1995-01-01'
and l_receiptdate < dateadd(year,
+1, '1995-01-01')
group by
l_shipmode
order by
l_shipmode;
go
insert into TIMES values
('Q12 in Str01 end',getdate());

insert into TIMES values
('Q16 in Str01 start',getdate());
select
p_brand,
p_type,
p_size,
count(distinct ps_suppkey)
as supplier_cnt
from
partsupp,
part
where
p_partkey = ps_partkey
and p_brand <> 'Brand#50'
and p_type not like 'SMALL PLATED%'
and p_size
in (33, 48, 23, 43, 28, 49, 3, 14)
and ps_suppkey not in (
select
s_suppkey
from
supplier
where
s_comment
like '%Customer%Complaints%'
)
group by
p_brand,
p_type,
p_size
order by
supplier_cnt desc,
p_brand,
p_type,
p_size;
go
insert into TIMES values
('Q16 in Str01 end',getdate());

insert into TIMES values
('Q15 in Str01 start',getdate());
select
s_suppkey,
s_name,
s_address,
s_phone,
total_revenue
from
supplier,
revenue1
where
s_suppkey = supplier_no
and total_revenue = (
select
max(total_revenue)
from
revenue1
)
order by
s_suppkey;
go
drop view revenue1;
go
insert into TIMES values
('Q15 in Str01 end',getdate());

insert into TIMES values
('Q13 in Str01 start',getdate());
select
c_count,
count(*) as custdist
from
(
select
c_custkey,
count(o_orderkey)
from
customer left outer join orders on
c_custkey = o_custkey
and o_comment not
like '%unusual%requests%'
group by
c_custkey
) as c_orders (c_custkey, c_count)
group by
c_count
order by
custdist desc,
c_count desc;
go
insert into TIMES values
('Q13 in Str01 end',getdate());

set rowcount 20
go

insert into TIMES values
('Q10 in Str01 start',getdate());
select
c_custkey,
c_name,
sum(l_extendedprice * (1 - l_discount))
as revenue,
c_acctbal,
n_name,
c_address,
c_phone,
c_comment
from
customer,
orders,
lineitem,
nation
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate >=  '1994-10-01'
and o_orderdate < dateadd(month,
+3, '1994-10-01')
and l_returnflag = 'R'
and c_nationkey = n_nationkey
group by
c_custkey,
c_name,
c_acctbal,
c_phone,
n_name,
c_address,
c_comment
order by
revenue desc;
go
insert into TIMES values
('Q10 in Str01 end',getdate());

set rowcount 100
go

insert into TIMES values
('Q2 in Str01 start',getdate());
select
s_acctbal,
s_name,
n_name,
p_partkey,
p_mfgr,
s_address,
s_phone,
s_comment
from
part,
supplier,
partsupp,
nation,
```

```
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and p_size = 15
and p_type like '%COPPER'
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'ASIA'
and ps_supplycost = (
select
min(ps_supplycost)
from
partsupp,
supplier,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'ASIA'
)
order by
s_acctbal desc,
n_name,
s_name,
p_partkey;
go
insert into TIMES values
('Q2 in Str01 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q8 in Str01 start',getdate());
select
o_year,
sum(case
when nation = 'KENYA' then volume
else 0
end) / sum(volume) as mkt_share
from
(
select
datepart(year, o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
as volume,
n2.n_name as nation
from
part,
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2,
region
where
p_partkey = l_partkey
and s_suppkey = l_suppkey
and l_orderkey = o_orderkey
and o_custkey = c_custkey
and c_nationkey = n1.n_nationkey
and n1.n_regionkey = r_regionkey
and r_name = 'AFRICA'
and s_nationkey = n2.n_nationkey
and o_orderdate between  '1995-01-01'
and  '1996-12-31'
and p_type = 'STANDARD BRUSHED COPPER'
) as all_nations
group by
o_year
order by
o_year;
go
insert into TIMES values
('Q8 in Str01 end',getdate());

insert into TIMES values
('Q14 in Str01 start',getdate());
select
100.00 * sum(case
when p_type like 'PROMO%'
then l_extendedprice * (1 - l_discount)
else 0
end) / sum(l_extendedprice * (1 - l_discount))
as promo_revenue
from
lineitem,
part
where
l_partkey = p_partkey
and l_shipdate >=  '1995-08-01'
and l_shipdate < dateadd(month,
+1, '1995-08-01');
go
insert into TIMES values
('Q14 in Str01 end',getdate());

insert into TIMES values
('Q19 in Str01 start',getdate());
select
sum(l_extendedprice* (1 - l_discount))
as revenue
from
lineitem,
part
where
(
p_partkey = l_partkey
and p_brand = 'Brand#42'
and p_container
in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
and l_quantity >= 5 and l_quantity <= 5 + 10
and p_size between 1 and 5
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#43'
and p_container
in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
and l_quantity >= 18 and l_quantity <= 18 + 10
and p_size between 1 and 10
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#22'
and p_container
in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
and l_quantity >= 28 and l_quantity <= 28 + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
);
go
insert into TIMES values
('Q19 in Str01 end',getdate());

insert into TIMES values
('Q9 in Str01 start',getdate());
select
nation,
o_year,
sum(amount) as sum_profit
from
(
select
n_name as nation,
datepart(year, o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
- ps_supplycost * l_quantity as amount
from
part,
supplier,
lineitem,
partsupp,
orders,
nation
where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
```

```
and p_name like '%chocolate%'
) as profit
group by
nation,
o_year
order by
nation,
o_year desc;
go
insert into TIMES values
('Q9 in Str01 end',getdate());

insert into TIMES values
('Q22 in Str01 start',getdate());
select
cntrycode,
count(*) as numcust,
sum(c_acctbal) as totacctbal
from
(
select
substring(c_phone, 1, 2) as cntrycode,
c_acctbal
from
customer
where
substring(c_phone, 1, 2) in
('31', '14', '19', '23', '33', '28', '27')
and c_acctbal > (
select
avg(c_acctbal)
from
customer
where
c_acctbal > 0.00
and substring(c_phone, 1, 2) in
('31', '14', '19', '23', '33', '28', '27')
)
and not exists (
select
*
from
orders
where
o_custkey = c_custkey
)
) as custsale
group by
cntrycode
order by
cntrycode;
go
insert into TIMES values
('Q22 in Str01 end',getdate());

insert into TIMES values
('Q1 in Str01 start',getdate());
select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount))
as sum_disc_price,
sum(l_extendedprice * (1 - l_discount)
* (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= dateadd(day,
-74, '1998-12-01')
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
go
insert into TIMES values
('Q1 in Str01 end',getdate());

insert into TIMES values
('Q4 in Str01 start',getdate());
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >=  '1995-11-01'
and o_orderdate < dateadd(month,
+3, '1995-11-01')
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
go
insert into TIMES values
('Q4 in Str01 end',getdate());
go
insert into TIMES values
('Str01 end',getdate());
go
```

Stream02.sql

```
-- using 1027173121 as a seed to the RNG

create view revenue2
(supplier_no, total_revenue) as
select
l_suppkey,
sum(l_extendedprice * (1 - l_discount))
from
lineitem
where
l_shipdate >=  '1996-05-01'
and l_shipdate <  dateadd(month,
+3, '1996-05-01')
group by
l_suppkey;
go

insert into TIMES values
('Str02 start',getdate());

insert into TIMES values
('Q6 in Str02 start',getdate());
select
sum(l_extendedprice * l_discount)
as revenue
from
lineitem
where
l_shipdate >=  '1994-01-01'
and l_shipdate <  dateadd(year,
+1, '1994-01-01')
and l_discount between 0.09 - 0.01
and 0.09 + 0.01
and l_quantity < 24;
go
insert into TIMES values
('Q6 in Str02 end',getdate());

insert into TIMES values
('Q17 in Str02 start',getdate());
select
sum(l_extendedprice) / 7.0 as avg_yearly
from
lineitem,
part
where
p_partkey = l_partkey
and p_brand = 'Brand#53'
and p_container = 'JUMBO PKG'
and l_quantity < (
select
0.2 * avg(l_quantity)
from
```

```
lineitem
where
l_partkey = p_partkey
);
go
insert into TIMES values
('Q17 in Str02 end',getdate());

insert into TIMES values
('Q14 in Str02 start',getdate());
select
100.00 * sum(case
when p_type like 'PROMO%'
then l_extendedprice * (1 - l_discount)
else 0
end) / sum(l_extendedprice * (1 - l_discount))
'as promo_revenue
from
lineitem,
part
where
l_partkey = p_partkey
and l_shipdate >=  '1995-03-01'
and l_shipdate <  dateadd(month,
'+1, '1995-03-01');
go
insert into TIMES values
('Q14 in Str02 end',getdate());

insert into TIMES values
('Q16 in Str02 start',getdate());
select
p_brand,
p_type,
p_size,
count(distinct ps_suppkey) as supplier_cnt
from
partsupp,
part
where
p_partkey = ps_partkey
and p_brand <> 'Brand#30'
and p_type not like 'LARGE POLISHED%'
and p_size in (7, 23, 19, 11, 10, 41, 48, 44)
and ps_suppkey not in (
select
s_suppkey
from
supplier
where
s_comment like '%Customer%Complaints%'
)
group by
p_brand,
p_type,
p_size
order by
supplier_cnt desc,
p_brand,
p_type,
p_size;
go
insert into TIMES values
('Q16 in Str02 end',getdate());

insert into TIMES values
('Q19 in Str02 start',getdate());
select
sum(l_extendedprice* (1 - l_discount))
as revenue
from
lineitem,
part
where
(
p_partkey = l_partkey
and p_brand = 'Brand#41'
and p_container
in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
and l_quantity >= 1 and l_quantity <= 1 + 10
and p_size between 1 and 5
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#21'
and p_container
in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
and l_quantity >= 19 and l_quantity <= 19 + 10
and p_size between 1 and 10
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#11'
and p_container
in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
and l_quantity >= 24 and l_quantity <= 24 + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
);
go
insert into TIMES values
('Q19 in Str02 end',getdate());

set rowcount 20
go

insert into TIMES values
('Q10 in Str02 start',getdate());
select
c_custkey,
c_name,
sum(l_extendedprice * (1 - l_discount))
as revenue,
c_acctbal,
n_name,
c_address,
c_phone,
c_comment
from
customer,
orders,
lineitem,
nation
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate >=  '1994-08-01'
and o_orderdate <  dateadd(month,
+3, '1994-08-01')
and l_returnflag = 'R'
and c_nationkey = n_nationkey
group by
c_custkey,
c_name,
c_acctbal,
c_phone,
n_name,
c_address,
c_comment
order by
revenue desc;
go
insert into TIMES values
('Q10 in Str02 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q9 in Str02 start',getdate());
select
nation,
o_year,
sum(amount) as sum_profit
from
(
select
n_name as nation,
datepart(year, o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
- ps_supplycost * l_quantity as amount
from
part,
supplier,
lineitem,
partsupp,
orders,
```

```
nation
where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like '%blush%'
) as profit
group by
nation,
o_year
order by
nation,
o_year desc;
go
insert into TIMES values
('Q9 in Str02 end',getdate());

set rowcount 100
go

insert into TIMES values
('Q2 in Str02 start',getdate());
select
s_acctbal,
s_name,
n_name,
p_partkey,
p_mfgr,
s_address,
s_phone,
s_comment
from
part,
supplier,
partsupp,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and p_size = 3
and p_type like '%STEEL'
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'AFRICA'
and ps_supplycost = (
select
min(ps_supplycost)
from
partsupp,
supplier,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'AFRICA'
)
order by
s_acctbal desc,
n_name,
s_name,
p_partkey;
go
insert into TIMES values
('Q2 in Str02 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q15 in Str02 start',getdate());

select
s_suppkey,
s_name,
s_address,
s_phone,
total_revenue
from
supplier,
revenue2
where
s_suppkey = supplier_no
and total_revenue = (
select
max(total_revenue)
from
revenue2
)
order by
s_suppkey;
go
drop view revenue2;
go
insert into TIMES values
('Q15 in Str02 end',getdate());

insert into TIMES values
('Q8 in Str02 start',getdate());
select
o_year,
sum(case
when nation = 'FRANCE' then volume
else 0
end) / sum(volume) as mkt_share
from
(
select
datepart(year, o_orderdate) as o_year,
l_extendedprice * (1 - l_discount) as volume,
n2.n_name as nation
from
part,
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2,
region
where
p_partkey = l_partkey
and s_suppkey = l_suppkey
and l_orderkey = o_orderkey
and o_custkey = c_custkey
and c_nationkey = n1.n_nationkey
and n1.n_regionkey = r_regionkey
and r_name = 'EUROPE'
and s_nationkey = n2.n_nationkey
and o_orderdate between  '1995-01-01'
and  '1996-12-31'
and p_type = 'STANDARD POLISHED TIN'
) as all_nations
group by
o_year
order by
o_year;
go
insert into TIMES values
('Q8 in Str02 end',getdate());

insert into TIMES values
('Q5 in Str02 start',getdate());
select
n_name,
sum(l_extendedprice * (1 - l_discount))
as revenue
from
customer,
orders,
lineitem,
supplier,
nation,
region
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'EUROPE'
and o_orderdate >=  '1994-01-01'
and o_orderdate <  dateadd(year,
+1, '1994-01-01')
group by
n_name
order by
```

```
revenue desc;
go
insert into TIMES values
('Q5 in Str02 end',getdate());

insert into TIMES values
('Q22 in Str02 start',getdate());
select
cntrycode,
count(*) as numcust,
sum(c_acctbal) as totacctbal
from
(
select
substring(c_phone, 1, 2) as cntrycode,
c_acctbal
from
customer
where
substring(c_phone, 1, 2) in
('29', '14', '30', '28', '31', '19', '33')
and c_acctbal > (
select
avg(c_acctbal)
from
customer
where
c_acctbal > 0.00
and substring(c_phone, 1, 2) in
('29', '14', '30', '28', '31', '19', '33')
)
and not exists (
select
*
from
orders
where
o_custkey = c_custkey
)
) as custsale
group by
cntrycode
order by
cntrycode;
go
insert into TIMES values
('Q22 in Str02 end',getdate());

insert into TIMES values
('Q12 in Str02 start',getdate());
select
l_shipmode,
sum(case
when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1
else 0
end) as high_line_count,
sum(case
when o_orderpriority <> '1-URGENT'
and o_orderpriority <> '2-HIGH'
then 1
else 0
end) as low_line_count
from
orders,
lineitem
where
o_orderkey = l_orderkey
and l_shipmode in ('REG AIR', 'MAIL')
and l_commitdate < l_receiptdate
and l_shipdate < l_commitdate
and l_receiptdate >=  '1995-01-01'
and l_receiptdate <  dateadd(year,
+1, '1995-01-01')
group by
l_shipmode
order by
l_shipmode;
go
insert into TIMES values
('Q12 in Str02 end',getdate());

insert into TIMES values
('Q7 in Str02 start',getdate());
select
supp_nation,
cust_nation,
l_year,
sum(volume) as revenue
from
(
select
n1.n_name as supp_nation,
n2.n_name as cust_nation,
datepart(year, l_shipdate) as l_year,
l_extendedprice * (1 - l_discount)
as volume
from
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2
where
s_suppkey = l_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and (
(n1.n_name = 'UNITED STATES'
and n2.n_name = 'FRANCE')
or (n1.n_name = 'FRANCE'
and n2.n_name = 'UNITED STATES')
)
and l_shipdate between  '1995-01-01'
and  '1996-12-31'
) as shipping
group by
supp_nation,
cust_nation,
l_year
order by
supp_nation,
cust_nation,
l_year;
go
insert into TIMES values
('Q7 in Str02 end',getdate());

insert into TIMES values
('Q13 in Str02 start',getdate());
select
c_count,
count(*) as custdist
from
(
select
c_custkey,
count(o_orderkey)
from
customer left outer join orders on
c_custkey = o_custkey
and o_comment not
like '%unusual%accounts%'
group by
c_custkey
) as c_orders (c_custkey, c_count)
group by
c_count
order by
custdist desc,
c_count desc;
go
insert into TIMES values
('Q13 in Str02 end',getdate());

set rowcount 100
go

insert into TIMES values
('Q18 in Str02 start',getdate());
select
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice,
sum(l_quantity)
from
customer,
orders,
```

```
lineitem
where
o_orderkey in (
select
l_orderkey
from
lineitem
group by
l_orderkey having
sum(l_quantity) > 314
)
and c_custkey = o_custkey
and o_orderkey = l_orderkey
group by
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice
order by
o_totalprice desc,
o_orderdate;
go
insert into TIMES values
('Q18 in Str02 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q1 in Str02 start',getdate());
select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount))
as sum_disc_price,
sum(l_extendedprice * (1 - l_discount)
* (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <=  dateadd(day,
-82, '1998-12-01')
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
go
insert into TIMES values
('Q1 in Str02 end',getdate());

insert into TIMES values
('Q4 in Str02 start',getdate());
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >=  '1993-05-01'
and o_orderdate <  dateadd(month,
+3, '1993-05-01')
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
go
insert into TIMES values
('Q4 in Str02 end',getdate());

insert into TIMES values
('Q20 in Str02 start',getdate());
select
s_name,
s_address
from
supplier,
nation
where
s_suppkey in (
select
ps_suppkey
from
partsupp
where
ps_partkey in (
select
p_partkey
from
part
where
p_name like 'goldenrod%'
)
and ps_availqty > (
select
0.5 * sum(l_quantity)
from
lineitem
where
l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and l_shipdate >=  '1994-01-01'
and l_shipdate <  dateadd(year,
+1, '1994-01-01')
)
)
and s_nationkey = n_nationkey
and n_name = 'IRAN'
order by
s_name;
go
insert into TIMES values
('Q20 in Str02 end',getdate());

set rowcount 10
go

insert into TIMES values
('Q3 in Str02 start',getdate());
select
l_orderkey,
sum(l_extendedprice * (1 - l_discount))
as revenue,
o_orderdate,
o_shippriority
from
customer,
orders,
lineitem
where
c_mktsegment = 'FURNITURE'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate <  '1995-03-01'
and l_shipdate >  '1995-03-01'
group by
l_orderkey,
o_orderdate,
o_shippriority
order by
revenue desc,
o_orderdate;
go
insert into TIMES values
('Q3 in Str02 end',getdate());

set rowcount 0
go

insert into TIMES values
('Q11 in Str02 start',getdate());
select
ps_partkey,
sum(ps_supplycost * ps_availqty) as value
from
partsupp,
supplier,
```

```
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'UNITED STATES'
group by
ps_partkey having
sum(ps_supplycost * ps_availqty) > (
select
sum(ps_supplycost * ps_availqty)
* 0.0001000000
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'UNITED STATES'
)
order by
value desc;
go
insert into TIMES values
('Q11 in Str02 end',getdate());

set rowcount 100
go

insert into TIMES values
('Q21 in Str02 start',getdate());
select
s_name,
count(*) as numwait
from
supplier,
lineitem l1,
orders,
nation
where
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (
select
*
from
lineitem l2
where
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (
select
*
from
lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
and n_name = 'INDONESIA'
group by
s_name
order by
numwait desc,
s_name;
go
insert into TIMES values
('Q21 in Str02 end',getdate());

set rowcount 0
go

insert into TIMES values
('Str02 end',getdate());
go
```

A.4 Load Test

load.cmd

```
rem Load Test starts

cd SQL_Server_Files
set cwd=%cd%


sqlcmd -E -dTPCH -Q"create table LOADTIMES
(STEP char(35),TIMESTAMP datetime)"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('LOAD begin',getdate())"

rem Creating the tables

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('Create Tables begin',getdate())"
sqlcmd -E -e -dTPCH -iCreateTables.sql
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('Create Tables end',getdate())"


rem Executing Bulk Inserts

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('NATION bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert NATION
from '%cd%\nation.tbl'
with (FieldTerminator = '|',
RowTerminator ='|\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('NATION end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('REGION bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert REGION
from '%cd%\region.tbl'
with (FieldTerminator = '|',
RowTerminator ='|\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('REGION end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('SUPPLIER bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert SUPPLIER
from '%cd%\supplier.tbl'
with (FieldTerminator = '|',
RowTerminator ='|\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('SUPPLIER end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('PART bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert PART
from '%cd%\part.tbl'
with (FieldTerminator = '|',
RowTerminator ='|\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('PART end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('PARTSUPP bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert PARTSUPP
from '%cd%\partsupp.tbl'
with (FieldTerminator = '|',
RowTerminator ='|\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('PARTSUPP end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('ORDERS bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert ORDERS
from '%cd%\orders.tbl'
with (FieldTerminator = '|',
RowTerminator ='|\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('ORDERS end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('LINEITEM bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert LINEITEM
from '%cd%\lineitem.tbl'
with (FieldTerminator = '|',
```

```
RowTerminator ='|\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('LINEITEM end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('CUSTOMER bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert CUSTOMER
from '%cd%\customer.tbl'
with (FieldTerminator = '|',
RowTerminator ='|\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('CUSTOMER end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('NEWLINEITEM bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert NEWLINEITEM
from '%cd%\Lineitem.tbl.u1'
with (FieldTerminator = '|',
RowTerminator ='\n',tablock)"
sqlcmd -E -e -dTPCH -Q"bulk insert NEWLINEITEM
from '%cd%\Lineitem.tbl.u2'
with (FieldTerminator = '|',
RowTerminator ='\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('NEWLINEITEM end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('NEWORDERS bulk insert begin',getdate())"
sqlcmd -E -e -dTPCH -Q"bulk insert NEWORDERS
from '%cd%\Orders.tbl.u1'
with (FieldTerminator = '|',
RowTerminator ='\n',tablock)"
sqlcmd -E -e -dTPCH -Q"bulk insert NEWORDERS
from '%cd%\Orders.tbl.u2'
with (FieldTerminator = '|',
RowTerminator ='\n',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('NEWORDERS end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('OLDORDERS bulk insert begin',getdate())"
sqlcmd -E -dTPCH -Q"bulk insert OLDORDERS
from '%cd%\Delete.1'
with (FieldTerminator = '|',
RowTerminator ='|',tablock)"
sqlcmd -E -dTPCH -Q"bulk insert OLDORDERS
from '%cd%\Delete.2'
with (FieldTerminator = '|',
RowTerminator ='|',tablock)"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('OLDORDERS end',getdate())"


rem Creating Constraints

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('Creating constraints begin',getdate())"
sqlcmd -E -e -dTPCH -iConstraints.sql
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('Creating constraints end',getdate())"


rem Creating Indexes

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('Creating indexes begin',getdate())"
sqlcmd -E -e -dTPCH -iIndexes.sql
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('Creating indexes end',getdate())"


rem Collecting Statistics

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('Creating statistics start',getdate())"
sqlcmd -E -e -dTPCH -Q"sp_createstats"
sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('Creating statistics end',getdate())"


rem Installing Refresh Functions
as Stored Procedures

sqlcmd -E -dTPCH -Q"insert into LOADTIMES values
('Installing refresh functions start',getdate())"
sqlcmd -E -e -I -dTPCH -iCreateRF1.sql
sqlcmd -E -e -I -dTPCH -iCreateRF2.sql

sqlcmd -E -dTPCH -Q"insert into LOADTIMES values
('Installing refresh functions end',getdate())"

sqlcmd -E -dTPCH -Q"insert into LOADTIMES
values ('LOAD end',getdate())"


cd ..
semaphore -release SEM1
semaphore -release SEM1

exit /B
```

A.5 Performance Test

run.cmd

```
cd SQL_Server_Files


rem Power Test

sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Power start',getdate())"

sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str00 RF1 start',getdate())"
OSQL -I -E -dTPCH -Q"exec RF1 40"
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str00 RF1 end',getdate())"


sqlcmd -I -E -dTPCH -iStream00.sql


sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str00 RF2 start',getdate())"
OSQL -I -E -dTPCH -Q"exec RF2 32"
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str00 RF2 end',getdate())"


sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Power end',getdate())"



rem Throughput Test

sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Throughput start',getdate())"

cd ..
start cmd /C RunStream01.cmd
start cmd /C RunStream02.cmd
semaphore -wait SEM2


cd SQL_Server_Files
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str01 RF1 start',getdate())"
OSQL -I -E -dTPCH -Q"exec RF1 4008"
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str01 RF1 end',getdate())"
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str01 RF2 start',getdate())"
OSQL -I -E -dTPCH -Q"exec RF2 4000"
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str01 RF2 end',getdate())"

sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str02 RF1 start',getdate())"
OSQL -I -E -dTPCH -Q"exec RF1 7976"
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str02 RF1 end',getdate())"
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str02 RF2 start',getdate())"
OSQL -I -E -dTPCH -Q"exec RF2 7968"
sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Str02 RF2 end',getdate())"

sqlcmd -I -E -dTPCH -Q"insert into TIMES
values ('Throughput end',getdate())"
```

```
cd ..
semaphore -release SEM1
semaphore -release SEM1

exit /B
```

RunStream01.cmd

```
cd SQL_Server_Files
sqlcmd -I -E -dTPCH -iStream01.sql

cd ..
semaphore -release SEM2

exit /B
```

RunStream02.cmd

```
cd SQL_Server_Files
sqlcmd -I -E -dTPCH -iStream02.sql

cd ..
semaphore -release SEM2

exit /B
```

A.6 Full Test

all_tests.cmd

```
rem Test start

sqlcmd -I -E -Q"drop database TPCH"
sqlcmd -I -E -Q"CHECKPOINT"
sqlcmd -I -E -Q"DBCC FREEPROCCACHE"
sqlcmd -I -E -Q"DBCC DROPCLEANBUFFERS"

net stop mssqlserver

rem Starting the server

net start mssqlserver

cd SQL_Server_Files


rem Creating database TPCH

sqlcmd -I -E -Q"create database TPCH"


cd ..

start cmd /C load.cmd

semaphore -wait SEM1


sqlcmd -I -E -dTPCH -Q"select * from loadtimes
order by timestamp"
-o SQL_Server_Results\load.txt -b

start cmd /C run.cmd

semaphore -wait SEM1


sqlcmd -I -E -dTPCH -Q"select * from times
order by start"
-o SQL_Server_Results\performance.txt -b


sqlcmd -I -E -Q"drop database TPCH".

sqlcmd -I -E -Q"CHECKPOINT"

sqlcmd -I -E -Q"DBCC FREEPROCCACHE"

sqlcmd -I -E -Q"DBCC DROPCLEANBUFFERS"

net stop mssqlserver

exit \B
```

A.7 Concurrency Handling

semaphore.cpp

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <string.h>
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

int main(int argc, char **argv)
{
typedef enum {eUnknown, eStart,
eWait, eRelease} OPERATION;
OPERATION eOP = eUnknown;
int i;
HANDLE hSemaphore;

if (_stricmp(argv[1], "-wait") == 0)
eOP = eWait;
else if (_stricmp(argv[1], "-release") == 0)
eOP = eRelease;

if (eOP == eWait)
{
hSemaphore = CreateSemaphore(NULL, 0,
2000000000,argv[2]);
for (i=0; i<2; i++)
{
WaitForSingleObject(hSemaphore, INFINITE);
}
CloseHandle(hSemaphore);
}

else if (eOP == eRelease)
{
hSemaphore = OpenSemaphore
(SEMAPHORE_MODIFY_STATE, FALSE, argv[2]);
ReleaseSemaphore(hSemaphore, 1, NULL);
CloseHandle(hSemaphore);
}

return 0;
}
```

Appendix B

Source code for MySQL 5.1

B.1 Database Build Scripts

CreateTables.sql

```
create table LOADTIMES
(STEP char(35),TIMESTAMP datetime);

insert into LOADTIMES values
('LOAD begin',NOW());

insert into LOADTIMES values
('Create Tables begin',NOW());

create table REGION
(R_REGIONKEY int not null,
R_NAME char(25) not null,
R_COMMENT varchar(152) not null,
PRIMARY KEY (R_REGIONKEY));

create table NATION
(N_NATIONKEY int not null,
N_NAME char(25) not null,
N_REGIONKEY int not null,
N_COMMENT varchar(152) not null,
PRIMARY KEY (N_NATIONKEY),
INDEX (N_REGIONKEY));

create table SUPPLIER
(S_SUPPKEY int not null,
S_NAME char(25) not null,
S_ADDRESS varchar(40) not null,
S_NATIONKEY int not null,
S_PHONE char(15) not null,
S_ACCTBAL float not null,
S_COMMENT varchar(101) not null,
PRIMARY KEY (S_SUPPKEY),
INDEX (S_NATIONKEY));

create table CUSTOMER
(C_CUSTKEY int not null,
C_NAME varchar(25) not null,
C_ADDRESS varchar(40) not null,
C_NATIONKEY int not null,
C_PHONE char(15) not null,
C_ACCTBAL float not null,
C_MKTSEGMENT char(10) not null,
C_COMMENT varchar(117) not null,
PRIMARY KEY (C_CUSTKEY),
INDEX (C_NATIONKEY));

create table PART
(P_PARTKEY int not null,
P_NAME varchar(55) not null,
P_MFGR char(25) not null,
P_BRAND char(10) not null,
P_TYPE varchar(25) not null,
P_SIZE int not null,
P_CONTAINER char(10) not null,
P_RETAILPRICE float not null,
P_COMMENT varchar(23) not null,
PRIMARY KEY (P_PARTKEY));

create table PARTSUPP
(PS_PARTKEY int not null,
PS_SUPPKEY int not null,
PS_AVAILQTY int not null,
PS_SUPPLYCOST float not null,
PS_COMMENT varchar(199) not null,
PRIMARY KEY (PS_PARTKEY,PS_SUPPKEY),
INDEX (PS_SUPPKEY),
INDEX (PS_PARTKEY));

create table ORDERS
(O_ORDERKEY bigint not null,
O_CUSTKEY int not null,
O_ORDERSTATUS char(1) not null,
O_TOTALPRICE float not null,
O_ORDERDATE date not null,
O_ORDERPRIORITY char(15) not null,
O_CLERK char(15) not null,
O_SHIPPRIORITY int not null,
O_COMMENT varchar(79) not null,
PRIMARY KEY (O_ORDERKEY),
INDEX (O_CUSTKEY),
INDEX (O_ORDERDATE));

create table LINEITEM
(L_ORDERKEY bigint not null,
L_PARTKEY int not null,
L_SUPPKEY int not null,
L_LINENUMBER int not null,
L_QUANTITY float not null,
L_EXTENDEDPRICE float not null,
L_DISCOUNT float not null,
L_TAX float not null,
L_RETURNFLAG char(1) not null,
L_LINESTATUS char(1) not null,
L_SHIPDATE date not null,
L_COMMITDATE date not null,
L_RECEIPTDATE date not null,
L_SHIPINSTRUCT char(25) not null,
L_SHIPMODE char(10) not null,
L_COMMENT varchar(44) not null,
PRIMARY KEY (L_ORDERKEY,L_LINENUMBER),
INDEX (L_ORDERKEY),
INDEX (L_LINENUMBER),
INDEX (L_SHIPDATE),
INDEX (L_COMMITDATE),
INDEX (L_RECEIPTDATE));

create table TIMES
(QUERY char(25),
START datetime);

create table NEWORDERS
(O_ORDERKEY bigint not null,
O_CUSTKEY int not null,
O_ORDERSTATUS char(1) not null,
O_TOTALPRICE float not null,
O_ORDERDATE datetime not null,
O_ORDERPRIORITY char(15) not null,
```

```
O_CLERK char(15) not null,
O_SHIPPRIORITY int not null,
O_COMMENT varchar(79) not null);

create table NEWLINEITEM
(L_ORDERKEY bigint not null,
L_PARTKEY int not null,
L_SUPPKEY int not null,
L_LINENUMBER int not null,
L_QUANTITY float not null,
L_EXTENDEDPRICE float not null,
L_DISCOUNT float not null,
L_TAX float not null,
L_RETURNFLAG char(1) not null,
L_LINESTATUS char(1) not null,
L_SHIPDATE datetime not null,
L_COMMITDATE datetime not null,
L_RECEIPTDATE datetime not null,
L_SHIPINSTRUCT char(25) not null,
L_SHIPMODE char(10 ) not null,
L_COMMENT varchar(44) not null);

create table OLDORDERS
(O_ORDERKEY int not null);

insert into LOADTIMES values
('Create Tables end',NOW());
```

Inserts.sql

```
insert into LOADTIMES values
('NATION bulk insert begin',NOW());
LOAD DATA INFILE "nation.tbl"
INTO TABLE NATION FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('NATION end',NOW());

insert into LOADTIMES values
('REGION bulk insert begin',NOW());
LOAD DATA INFILE "region.tbl"
INTO TABLE REGION FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('REGION end',NOW());

insert into LOADTIMES values
('SUPPLIER bulk insert begin',NOW());
LOAD DATA INFILE "supplier.tbl"
INTO TABLE SUPPLIER FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('SUPPLIER end',NOW());

insert into LOADTIMES values
('PART bulk insert begin',NOW());
LOAD DATA INFILE "part.tbl"
INTO TABLE PART FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('PART end',NOW());

insert into LOADTIMES values
('PARTSUPP bulk insert begin',NOW());
LOAD DATA INFILE "partsupp.tbl"
INTO TABLE PARTSUPP FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('PARTSUPP end',NOW());

insert into LOADTIMES values
('ORDERS bulk insert begin',NOW());
LOAD DATA INFILE "orders.tbl"
INTO TABLE ORDERS FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('ORDERS end',NOW());

insert into LOADTIMES values
('LINEITEM bulk insert begin',NOW());
LOAD DATA INFILE "lineitem.tbl"
INTO TABLE LINEITEM FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('LINEITEM end',NOW());

insert into LOADTIMES values
('CUSTOMER bulk insert begin',NOW());
LOAD DATA INFILE "customer.tbl"
INTO TABLE CUSTOMER FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('CUSTOMER end',NOW());

insert into LOADTIMES values
('NEWLINEITEM bulk insert begin',NOW());
LOAD DATA INFILE "Lineitem.tbl.u1"
INTO TABLE NEWLINEITEM FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
LOAD DATA INFILE "Lineitem.tbl.u2"
INTO TABLE NEWLINEITEM FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('NEWLINEITEM end',NOW());

insert into LOADTIMES values
('NEWORDERS bulk insert begin',NOW());
LOAD DATA INFILE "Orders.tbl.u1"
INTO TABLE NEWORDERS FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
LOAD DATA INFILE "Orders.tbl.u2"
INTO TABLE NEWORDERS FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('NEWORDERS end',NOW());

insert into LOADTIMES values
('OLDORDERS bulk insert begin',NOW());
LOAD DATA INFILE "Delete.1"
INTO TABLE OLDORDERS FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
LOAD DATA INFILE "Delete.2"
INTO TABLE OLDORDERS FIELDS TERMINATED BY "|"
LINES TERMINATED BY "\r\n";
insert into LOADTIMES values
('OLDORDERS end',NOW());
```

CreateStatistics.sql

```
insert into LOADTIMES values
('Creating statistics start',NOW());

ANALYZE TABLE nation;
ANALYZE TABLE region;
ANALYZE TABLE supplier;
ANALYZE TABLE part;
ANALYZE TABLE partsupp;
ANALYZE TABLE orders;
ANALYZE TABLE lineitem;
ANALYZE TABLE customer;

insert into LOADTIMES values
('Creating statistics end',NOW());

insert into LOADTIMES values
('Installing refresh functions start',NOW());
```

LoadEnd.sql

```
use tpch;

insert into LOADTIMES values
('Installing refresh functions end',NOW());

insert into LOADTIMES values
('LOAD end',NOW());
```

B.2 Refresh Function Definitions

CreateRF1.sql

```
DELIMITER //
CREATE PROCEDURE RF1
(IN startkey INT)

BEGIN

DECLARE loops INT;

SET loops = 124;

WHILE loops > 0 DO
INSERT INTO ORDERS
(O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT)
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = startkey;

INSERT INTO LINEITEM
(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT)
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
WHERE L_ORDERKEY=O_ORDERKEY
AND O_ORDERKEY = startkey;

SET startkey = (startkey + 1);

INSERT INTO ORDERS
(O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT)
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = startkey;

INSERT INTO LINEITEM
(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT)
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
WHERE L_ORDERKEY=O_ORDERKEY
AND O_ORDERKEY = startkey;

SET startkey = (startkey + 1);

INSERT INTO ORDERS
(O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT)
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = startkey;

INSERT INTO LINEITEM
(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT)
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
WHERE L_ORDERKEY=O_ORDERKEY
AND O_ORDERKEY = startkey;

SET startkey = (startkey + 1);

INSERT INTO ORDERS
(O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT)
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = startkey;

INSERT INTO LINEITEM
(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT)
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
WHERE L_ORDERKEY=O_ORDERKEY
AND O_ORDERKEY = startkey;

SET startkey = (startkey + 1);

INSERT INTO ORDERS
(O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT)
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = startkey;

INSERT INTO LINEITEM
(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT)
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
WHERE L_ORDERKEY=O_ORDERKEY
AND O_ORDERKEY = startkey;

SET startkey = (startkey + 1);

INSERT INTO ORDERS
(O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT)
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = startkey;

INSERT INTO LINEITEM
(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT)
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
```

```
WHERE L_ORDERKEY=O_ORDERKEY
AND O_ORDERKEY = startkey;

SET startkey = (startkey + 1);

INSERT INTO ORDERS
(O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT)
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = startkey;

INSERT INTO LINEITEM
(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT)
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
WHERE L_ORDERKEY=O_ORDERKEY
AND O_ORDERKEY = startkey;

SET startkey = (startkey + 1);

INSERT INTO ORDERS
(O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT)
SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
O_CLERK, O_SHIPPRIORITY, O_COMMENT
FROM NEWORDERS
WHERE O_ORDERKEY = startkey;

INSERT INTO LINEITEM
(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG,
L_LINESTATUS, L_SHIPDATE, L_COMMITDATE,
L_RECEIPTDATE, L_SHIPINSTRUCT,
L_SHIPMODE, L_COMMENT)
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
FROM NEWLINEITEM, NEWORDERS
WHERE L_ORDERKEY=O_ORDERKEY
AND O_ORDERKEY = startkey;

SET startkey = (startkey + 25);

SET loops = (loops - 1);

END WHILE;

END //
DELIMITER ;
GO
```

CreateRF2.sql

```
DELIMITER //

CREATE PROCEDURE RF2

(IN startkey INT)

BEGIN

DECLARE loops INT;

SET loops = 124;

WHILE loops > 0 DO
DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

SET startkey = (startkey + 1);

DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

SET startkey = (startkey + 1);

DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

SET startkey = (startkey + 1);

DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

SET startkey = (startkey + 1);

DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

SET startkey = (startkey + 1);

DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

SET startkey = (startkey + 1);

DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
```

```
WHERE O_ORDERKEY = startkey);

DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

SET startkey = (startkey + 1);

DELETE FROM ORDERS
WHERE O_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

DELETE FROM LINEITEM
WHERE L_ORDERKEY in
(SELECT O_ORDERKEY
FROM OLDORDERS
WHERE O_ORDERKEY = startkey);

SET startkey = (startkey + 25);

SET loops = (loops - 1);

END WHILE;

END //
GO
```

B.3 Query Streams

Stream00.sql

```
use tpch;

create view revenue0
(supplier_no, total_revenue) as
select
l_suppkey,
sum(l_extendedprice * (1 - l_discount))
from
lineitem
where
l_shipdate >= date '1996-04-01'
and l_shipdate < date '1996-04-01'
+interval '3' month
group by
l_suppkey;
go

insert into TIMES values
('Str00 start',NOW());


insert into TIMES values
('Q14 in Str00 start',NOW());
select
100.00 * sum(case
when p_type like 'PROMO%'
then l_extendedprice * (1 - l_discount)
else 0
end) / sum(l_extendedprice * (1 - l_discount))
as promo_revenue
from
lineitem,
part
where
l_partkey = p_partkey
and l_shipdate >= date '1995-08-01'
and l_shipdate < date '1995-08-01'
+interval '1' month
go
insert into TIMES values
('Q14 in Str00 end',NOW());

insert into TIMES values
('Q2 in Str00 start',NOW());
select
s_acctbal,
s_name,
n_name,
p_partkey,
p_mfgr,
s_address,
s_phone,
s_comment
from
part,
supplier,
partsupp,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and p_size = 27
and p_type like '%TIN'
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'MIDDLE EAST'
and ps_supplycost = (
select
min(ps_supplycost)
from
partsupp,
supplier,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'MIDDLE EAST'
)
order by
s_acctbal desc,
n_name,
s_name,
p_partkey
limit 100;
go
insert into TIMES values
('Q2 in Str00 end',NOW());

insert into TIMES values
('Q9 in Str00 start',NOW());
select
nation,
o_year,
sum(amount) as sum_profit
from
(
select
n_name as nation,
extract(year from o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
- ps_supplycost * l_quantity as amount
from
part,
supplier,
lineitem,
partsupp,
orders,
nation
where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like '%dark%'
) as profit
group by
nation,
o_year
order by
nation,
o_year desc;
go
insert into TIMES values
('Q9 in Str00 end',NOW());

insert into TIMES values
('Q20 in Str00 start',NOW());
```

```
select
s_name,
s_address
from
supplier,
nation
where
s_suppkey in (
select
ps_suppkey
from
partsupp
where
ps_partkey in (
select
p_partkey
from
part
where
p_name like 'maroon%'
)
and ps_availqty > (
select
0.5 * sum(l_quantity)
from
lineitem
where
l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and l_shipdate >= date '1997-01-01'
and l_shipdate < date '1997-01-01'
+interval '1' year
)
)
and s_nationkey = n_nationkey
and n_name = 'FRANCE'
order by
s_name;
go
insert into TIMES values
('Q20 in Str00 end',NOW());

insert into TIMES values
('Q6 in Str00 start',NOW());
select
sum(l_extendedprice * l_discount)
as revenue
from
lineitem
where
l_shipdate >= date '1993-01-01'
and l_shipdate < date '1993-01-01'
+interval '1' year
and l_discount between 0.06 - 0.01
and 0.06 + 0.01
and l_quantity < 25;
go
insert into TIMES values
('Q6 in Str00 end',NOW());

insert into TIMES values
('Q17 in Str00 start',NOW());
select
sum(l_extendedprice) / 7.0
as avg_yearly
from
lineitem,
part
where
p_partkey = l_partkey
and p_brand = 'Brand#52'
and p_container = 'MED DRUM'
and l_quantity < (
select
0.2 * avg(l_quantity)
from
lineitem
where
l_partkey = p_partkey
);
go
insert into TIMES values
('Q17 in Str00 end',NOW());

insert into TIMES values
('Q18 in Str00 start',NOW());
select
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice,
sum(l_quantity)
from
customer,
orders,
lineitem
where
o_orderkey in (
select
l_orderkey
from
lineitem
group by
l_orderkey having
sum(l_quantity) > 315
)
and c_custkey = o_custkey
and o_orderkey = l_orderkey
group by
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice
order by
o_totalprice desc,
o_orderdate
limit 100;
go
insert into TIMES values
('Q18 in Str00 end',NOW());

insert into TIMES values
('Q8 in Str00 start',NOW());
select
o_year,
sum(case
when nation = 'RUSSIA'
then volume
else 0
end) / sum(volume) as mkt_share
from
(
select
extract(year from o_orderdate)
as o_year,
l_extendedprice * (1 - l_discount)
as volume,
n2.n_name
as nation
from
part,
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2,
region
where
p_partkey = l_partkey
and s_suppkey = l_suppkey
and l_orderkey = o_orderkey
and o_custkey = c_custkey
and c_nationkey = n1.n_nationkey
and n1.n_regionkey = r_regionkey
and r_name = 'EUROPE'
and s_nationkey = n2.n_nationkey
and o_orderdate between
date '1995-01-01'
and date '1996-12-31'
and p_type = 'SMALL BURNISHED COPPER'
) as all_nations
group by
o_year
order by
o_year;
go
insert into TIMES values
('Q8 in Str00 end',NOW());

insert into TIMES values
('Q21 in Str00 start',NOW());
```

```
select
s_name,
count(*) as numwait
from
supplier,
lineitem l1,
orders,
nation
where
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (
select
*
from
lineitem l2
where
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (
select
*
from
lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
and n_name = 'UNITED STATES'
group by
s_name
order by
numwait desc,
s_name
limit 100;
go
insert into TIMES values
('Q21 in Str00 end',NOW());

insert into TIMES values
('Q13 in Str00 start',NOW());
select
c_count,
count(*) as custdist
from
(
select
c_custkey,
count(o_orderkey)
from
customer left outer join orders on
c_custkey = o_custkey
and o_comment not
like '%unusual%requests%'
group by
c_custkey
) as c_orders (c_custkey, c_count)
group by
c_count
order by
custdist desc,
c_count desc;
go
insert into TIMES values
('Q13 in Str00 end',NOW());

insert into TIMES values
('Q3 in Str00 start',NOW());
select
l_orderkey,
sum(l_extendedprice * (1 - l_discount))
as revenue,
o_orderdate,
o_shippriority
from
customer,
orders,
lineitem
where
c_mktsegment = 'FURNITURE'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-30'
and l_shipdate > date '1995-03-30'
group by
l_orderkey,
o_orderdate,
o_shippriority
order by
revenue desc,
o_orderdate
limit 10;
go
insert into TIMES values
('Q3 in Str00 end',NOW());

insert into TIMES values
('Q22 in Str00 start',NOW());
select
cntrycode,
count(*) as numcust,
sum(c_acctbal) as totacctbal
from
(
select
substring(c_phone, 1, 2) as cntrycode,
c_acctbal
from
customer
where
substring(c_phone, 1, 2) in
('16', '29', '33', '34', '26', '22', '31')
and c_acctbal > (
select
avg(c_acctbal)
from
customer
where
c_acctbal > 0.00
and substring(c_phone from 1 for 2) in
('16', '29', '33', '34', '26', '22', '31')
)
and not exists (
select
*
from
orders
where
o_custkey = c_custkey
)
) as custsale
group by
cntrycode
order by
cntrycode;
go
insert into TIMES values
('Q22 in Str00 end',NOW());

insert into TIMES values
('Q16 in Str00 start',NOW());
select
p_brand,
p_type,
p_size,
count(distinct ps_suppkey)
as supplier_cnt
from
partsupp,
part
where
p_partkey = ps_partkey
and p_brand <> 'Brand#10'
and p_type not like 'PROMO ANODIZED%'
and p_size in
(30, 27, 50, 23, 2, 33, 49, 15)
and ps_suppkey not in (
select
s_suppkey
from
supplier
where
s_comment
like '%Customer%Complaints%'
)
group by
p_brand,
p_type,
p_size
```

```
order by
supplier_cnt desc,
p_brand,
p_type,
p_size;
go
insert into TIMES values
('Q16 in Str00 end',NOW());

insert into TIMES values
('Q4 in Str00 start',NOW());
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >= date '1997-09-01'
and o_orderdate < date '1997-09-01'
+interval '3' month
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
go
insert into TIMES values
('Q4 in Str00 end',NOW());

insert into TIMES values
('Q11 in Str00 start',NOW());
select
ps_partkey,
sum(ps_supplycost * ps_availqty)
as value
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'UNITED KINGDOM'
group by
ps_partkey having
sum(ps_supplycost * ps_availqty) > (
select
sum(ps_supplycost * ps_availqty)
* 0.0001000000
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'UNITED KINGDOM'
)
order by
value desc;
go
insert into TIMES values
('Q11 in Str00 end',NOW());

insert into TIMES values
('Q15 in Str00 start',NOW());

select
s_suppkey,
s_name,
s_address,
s_phone,
total_revenue
from
supplier,
revenue0
where
s_suppkey = supplier_no
and total_revenue = (
select
max(total_revenue)
from
revenue0
)
order by
s_suppkey;
go
drop view revenue0;
go
insert into TIMES values
('Q15 in Str00 end',NOW());

insert into TIMES values
('Q1 in Str00 start',NOW());
select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount))
as sum_disc_price,
sum(l_extendedprice * (1 - l_discount)
* (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= date '1998-12-01'
-interval '66' day
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
go
insert into TIMES values
('Q1 in Str00 end',NOW());

insert into TIMES values
('Q10 in Str00 start',NOW());
select
c_custkey,
c_name,
sum(l_extendedprice
* (1 - l_discount)) as revenue,
c_acctbal,
n_name,
c_address,
c_phone,
c_comment
from
customer,
orders,
lineitem,
nation
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate >= date '1993-06-01'
and o_orderdate < date '1993-06-01'
+interval '3' month
and l_returnflag = 'R'
and c_nationkey = n_nationkey
group by
c_custkey,
c_name,
c_acctbal,
c_phone,
n_name,
c_address,
c_comment
order by
revenue desc
limit 20;
go
insert into TIMES values
('Q10 in Str00 end',NOW());

insert into TIMES values
('Q19 in Str00 start',NOW());
select
sum(l_extendedprice
```

```
* (1 - l_discount)) as revenue
from
lineitem,
part
where
(
p_partkey = l_partkey
and p_brand = 'Brand#31'
and p_container
in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
and l_quantity >= 10 and l_quantity <= 10 + 10
and p_size between 1 and 5
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#53'
and p_container
in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
and l_quantity >= 17 and l_quantity <= 17 + 10
and p_size between 1 and 10
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#24'
and p_container
in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
and l_quantity >= 20 and l_quantity <= 20 + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
);
go
insert into TIMES values
('Q19 in Str00 end',NOW());

insert into TIMES values
('Q5 in Str00 start',NOW());
select
n_name,
sum(l_extendedprice
* (1 - l_discount)) as revenue
from
customer,
orders,
lineitem,
supplier,
nation,
region
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'AMERICA'
and o_orderdate >= date '1993-01-01'
and o_orderdate < date '1993-01-01'
+interval '1' year
group by
n_name
order by
revenue desc;
go
insert into TIMES values
('Q5 in Str00 end',NOW());

insert into TIMES values
('Q7 in Str00 start',NOW());
select
supp_nation,
cust_nation,
l_year,
sum(volume) as revenue
from
(
select
n1.n_name as supp_nation,
n2.n_name as cust_nation,
extract(year from l_shipdate)
as l_year,
l_extendedprice * (1 - l_discount)
as volume
from
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2
where
s_suppkey = l_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and (
(n1.n_name = 'MOROCCO'
and n2.n_name = 'RUSSIA')
or (n1.n_name = 'RUSSIA'
and n2.n_name = 'MOROCCO')
)
and l_shipdate between
date '1995-01-01'
and date '1996-12-31'
) as shipping
group by
supp_nation,
cust_nation,
l_year
order by
supp_nation,
cust_nation,
l_year;
go
insert into TIMES values
('Q7 in Str00 end',NOW());

insert into TIMES values
('Q12 in Str00 start',NOW());
select
l_shipmode,
sum(case
when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1
else 0
end) as high_line_count,
sum(case
when o_orderpriority <> '1-URGENT'
and o_orderpriority <> '2-HIGH'
then 1
else 0
end) as low_line_count
from
orders,
lineitem
where
o_orderkey = l_orderkey
and l_shipmode in ('RAIL', 'MAIL')
and l_commitdate < l_receiptdate
and l_shipdate < l_commitdate
and l_receiptdate >= date '1995-01-01'
and l_receiptdate < date '1995-01-01'
+interval '1' year
group by
l_shipmode
order by
l_shipmode;
go
insert into TIMES values
('Q12 in Str00 end',NOW());
go
insert into TIMES values
('Str00 end',NOW());
go
```

Stream01.sql

```
use tpch;

create view revenue1
(supplier_no, total_revenue) as
select
l_suppkey,
sum(l_extendedprice * (1 - l_discount))
```

```
from
lineitem
where
l_shipdate >= date '1994-07-01'
and l_shipdate < date '1994-07-01'
+interval '3' month
group by
l_suppkey;
go

insert into TIMES values
('Str01 start',NOW());

insert into TIMES values
('Q21 in Str01 start',NOW());
select
s_name,
count(*) as numwait
from
supplier,
lineitem l1,
orders,
nation
where
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (
select
*
from
lineitem l2
where
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (
select
*
from
lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
and n_name = 'PERU'
group by
s_name
order by
numwait desc,
s_name
limit 100;
go
insert into TIMES values
('Q21 in Str01 end',NOW());

insert into TIMES values
('Q3 in Str01 start',NOW());
select
l_orderkey,
sum(l_extendedprice * (1 - l_discount))
as revenue,
o_orderdate,
o_shippriority
from
customer,
orders,
lineitem
where
c_mktsegment = 'MACHINERY'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-16'
and l_shipdate > date '1995-03-16'
group by
l_orderkey,
o_orderdate,
o_shippriority
order by
revenue desc,
o_orderdate
limit 10;
go
insert into TIMES values
('Q3 in Str01 end',NOW());

insert into TIMES values
('Q18 in Str01 start',NOW());
select
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice,
sum(l_quantity)
from
customer,
orders,
lineitem
where
o_orderkey in (
select
l_orderkey
from
lineitem
group by
l_orderkey having
sum(l_quantity) > 312
)
and c_custkey = o_custkey
and o_orderkey = l_orderkey
group by
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice
order by
o_totalprice desc,
o_orderdate
limit 100;
go
insert into TIMES values
('Q18 in Str01 end',NOW());

insert into TIMES values
('Q5 in Str01 start',NOW());
select
n_name,
sum(l_extendedprice * (1 - l_discount))
as revenue
from
customer,
orders,
lineitem,
supplier,
nation,
region
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'ASIA'
and o_orderdate >= date '1994-01-01'
and o_orderdate < date '1994-01-01'
+interval '1' year
group by
n_name
order by
revenue desc;
go
insert into TIMES values
('Q5 in Str01 end',NOW());

insert into TIMES values
('Q11 in Str01 start',NOW());
select
ps_partkey,
sum(ps_supplycost * ps_availqty) as value
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'IRAQ'
group by
```

```
ps_partkey having
sum(ps_supplycost * ps_availqty) > (
select
sum(ps_supplycost * ps_availqty)
* 0.0001000000
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'IRAQ'
)
order by
value desc;
go
insert into TIMES values
('Q11 in Str01 end',NOW());

insert into TIMES values
('Q7 in Str01 start',NOW());
select
supp_nation,
cust_nation,
l_year,
sum(volume) as revenue
from
(
select
n1.n_name as supp_nation,
n2.n_name as cust_nation,
extract(year from l_shipdate) as l_year,
l_extendedprice * (1 - l_discount) as volume
from
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2
where
s_suppkey = l_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and (
(n1.n_name = 'GERMANY'
and n2.n_name = 'KENYA')
or (n1.n_name = 'KENYA'
and n2.n_name = 'GERMANY')
)
and l_shipdate between
date '1995-01-01'
and date '1996-12-31'
) as shipping
group by
supp_nation,
cust_nation,
l_year
order by
supp_nation,
cust_nation,
l_year;
go
insert into TIMES values
('Q7 in Str01 end',NOW());

insert into TIMES values
('Q6 in Str01 start',NOW());
select
sum(l_extendedprice * l_discount)
as revenue
from
lineitem
where
l_shipdate >= date '1994-01-01'
and l_shipdate < date '1994-01-01'
+interval '1' year
and l_discount between 0.04 - 0.01
and 0.04 + 0.01
and l_quantity < 25;
go
insert into TIMES values
('Q6 in Str01 end',NOW());

insert into TIMES values
('Q20 in Str01 start',NOW());
select
s_name,
s_address
from
supplier,
nation
where
s_suppkey in (
select
ps_suppkey
from
partsupp
where
ps_partkey in (
select
p_partkey
from
part
where
p_name like 'tomato%'
)
and ps_availqty > (
select
0.5 * sum(l_quantity)
from
lineitem
where
l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and l_shipdate >= date '1996-01-01'
and l_shipdate < date '1996-01-01'
+interval '1' year
)
)
and s_nationkey = n_nationkey
and n_name = 'VIETNAM'
order by
s_name;
go
insert into TIMES values
('Q20 in Str01 end',NOW());

insert into TIMES values
('Q17 in Str01 start',NOW());
select
sum(l_extendedprice) / 7.0
as avg_yearly
from
lineitem,
part
where
p_partkey = l_partkey
and p_brand = 'Brand#51'
and p_container = 'JUMBO BAG'
and l_quantity < (
select
0.2 * avg(l_quantity)
from
lineitem
where
l_partkey = p_partkey
);
go
insert into TIMES values
('Q17 in Str01 end',NOW());

insert into TIMES values
('Q12 in Str01 start',NOW());
select
l_shipmode,
sum(case
when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1
else 0
end) as high_line_count,
sum(case
when o_orderpriority <> '1-URGENT'
and o_orderpriority <> '2-HIGH'
then 1
else 0
end) as low_line_count
from
orders,
lineitem
```

```
where
o_orderkey = l_orderkey
and l_shipmode in ('AIR', 'MAIL')
and l_commitdate < l_receiptdate
and l_shipdate < l_commitdate
and l_receiptdate >= date '1995-01-01'
and l_receiptdate < date '1995-01-01'
+interval '1' year
group by
l_shipmode
order by
l_shipmode;
go
insert into TIMES values
('Q12 in Str01 end',NOW());

insert into TIMES values
('Q16 in Str01 start',NOW());
select
p_brand,
p_type,
p_size,
count(distinct ps_suppkey)
as supplier_cnt
from
partsupp,
part
where
p_partkey = ps_partkey
and p_brand <> 'Brand#50'
and p_type not like 'SMALL PLATED%'
and p_size
in (33, 48, 23, 43, 28, 49, 3, 14)
and ps_suppkey not in (
select
s_suppkey
from
supplier
where
s_comment like '%Customer%Complaints%'
)
group by
p_brand,
p_type,
p_size
order by
supplier_cnt desc,
p_brand,
p_type,
p_size;
go
insert into TIMES values
('Q16 in Str01 end',NOW());

insert into TIMES values
('Q15 in Str01 start',NOW());
select
s_suppkey,
s_name,
s_address,
s_phone,
total_revenue
from
supplier,
revenue1
where
s_suppkey = supplier_no
and total_revenue = (
select
max(total_revenue)
from
revenue1
)
order by
s_suppkey;

drop view revenue1;
go
insert into TIMES values
('Q15 in Str01 end',NOW());

insert into TIMES values
('Q13 in Str01 start',NOW());
select
c_count,
count(*) as custdist
from
(
select
c_custkey,
count(o_orderkey)
from
customer left outer join orders on
c_custkey = o_custkey
and o_comment not
like '%unusual%requests%'
group by
c_custkey
) as c_orders (c_custkey, c_count)
group by
c_count
order by
custdist desc,
c_count desc;
go
insert into TIMES values
('Q13 in Str01 end',NOW());

insert into TIMES values
('Q10 in Str01 start',NOW());
select
c_custkey,
c_name,
sum(l_extendedprice
* (1 - l_discount)) as revenue,
c_acctbal,
n_name,
c_address,
c_phone,
c_comment
from
customer,
orders,
lineitem,
nation
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate >= date '1994-10-01'
and o_orderdate < date '1994-10-01'
+interval '3' month
and l_returnflag = 'R'
and c_nationkey = n_nationkey
group by
c_custkey,
c_name,
c_acctbal,
c_phone,
n_name,
c_address,
c_comment
order by
revenue desc
limit 20;
go
insert into TIMES values
('Q10 in Str01 end',NOW());

insert into TIMES values
('Q2 in Str01 start',NOW());
select
s_acctbal,
s_name,
n_name,
p_partkey,
p_mfgr,
s_address,
s_phone,
s_comment
from
part,
supplier,
partsupp,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and p_size = 15
and p_type like '%COPPER'
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'ASIA'
and ps_supplycost = (
```

```
select
min(ps_supplycost)
from
partsupp,
supplier,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'ASIA'
)
order by
s_acctbal desc,
n_name,
s_name,
p_partkey
go
insert into TIMES values
('Q2 in Str01 end',NOW());

insert into TIMES values
('Q8 in Str01 start',NOW());
select
o_year,
sum(case
when nation = 'KENYA' then volume
else 0
end) / sum(volume) as mkt_share
from
(
select
extract(year from o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
as volume,
n2.n_name as nation
from
part,
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2,
region
where
p_partkey = l_partkey
and s_suppkey = l_suppkey
and l_orderkey = o_orderkey
and o_custkey = c_custkey
and c_nationkey = n1.n_nationkey
and n1.n_regionkey = r_regionkey
and r_name = 'AFRICA'
and s_nationkey = n2.n_nationkey
and o_orderdate between  '1995-01-01'
and  '1996-12-31'
and p_type = 'STANDARD BRUSHED COPPER'
) as all_nations
group by
o_year
order by
o_year;
go
insert into TIMES values
('Q8 in Str01 end',NOW());

insert into TIMES values
('Q14 in Str01 start',NOW());
select
100.00 * sum(case
when p_type like 'PROMO%'
then l_extendedprice * (1 - l_discount)
else 0
end) / sum(l_extendedprice * (1 - l_discount))
as promo_revenue
from
lineitem,
part
where
l_partkey = p_partkey
and l_shipdate >= date '1995-08-01'
and l_shipdate < date '1995-08-01'
+interval '1' month
go
insert into TIMES values
('Q14 in Str01 end',NOW());

insert into TIMES values
('Q19 in Str01 start',NOW());
select
sum(l_extendedprice
* (1 - l_discount)) as revenue
from
lineitem,
part
where
(
p_partkey = l_partkey
and p_brand = 'Brand#42'
and p_container
in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
and l_quantity >= 5 and l_quantity <= 5 + 10
and p_size between 1 and 5
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#43'
and p_container
in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
and l_quantity >= 18 and l_quantity <= 18 + 10
and p_size between 1 and 10
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#22'
and p_container
in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
and l_quantity >= 28 and l_quantity <= 28 + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
);
go
insert into TIMES values
('Q19 in Str01 end',NOW());

insert into TIMES values
('Q9 in Str01 start',NOW());
select
nation,
o_year,
sum(amount) as sum_profit
from
(
select
n_name as nation,
extract(year from o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
- ps_supplycost * l_quantity as amount
from
part,
supplier,
lineitem,
partsupp,
orders,
nation
where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like '%chocolate%'
) as profit
group by
nation,
o_year
order by
nation,
o_year desc;
go
insert into TIMES values
('Q9 in Str01 end',NOW());

insert into TIMES values
```

```
('Q22 in Str01 start',NOW());
select
cntrycode,
count(*) as numcust,
sum(c_acctbal) as totacctbal
from
(
select
substring(c_phone, 1, 2) as cntrycode,
c_acctbal
from
customer
where
substring(c_phone, 1, 2) in
('31', '14', '19', '23', '33', '28', '27')
and c_acctbal > (
select
avg(c_acctbal)
from
customer
where
c_acctbal > 0.00
and substring(c_phone from 1 for 2) in
('31', '14', '19', '23', '33', '28', '27')
)
and not exists (
select
*
from
orders
where
o_custkey = c_custkey
)
) as custsale
group by
cntrycode
order by
cntrycode;
go
insert into TIMES values
('Q22 in Str01 end',NOW());

insert into TIMES values
('Q1 in Str01 start',NOW());
select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount))
as sum_disc_price,
sum(l_extendedprice * (1 - l_discount)
* (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= date '1998-12-01'
-interval '74' day
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
go
insert into TIMES values
('Q1 in Str01 end',NOW());

insert into TIMES values
('Q4 in Str01 start',NOW());
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >= date '1995-11-01'
and o_orderdate < date '1995-11-01'
+interval '3' month
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
go
insert into TIMES values
('Q4 in Str01 end',NOW());
go
insert into TIMES values
('Str01 end',NOW());
go
```

Stream02.sql

```
use tpch;

create view revenue2
(supplier_no, total_revenue) as
select
l_suppkey,
sum(l_extendedprice * (1 - l_discount))
from
lineitem
where
l_shipdate >= date '1996-05-01'
and l_shipdate <  date '1996-05-01'
+interval '3' month
group by
l_suppkey;
go

insert into TIMES values
('Str02 start',NOW());

insert into TIMES values
('Q6 in Str02 start',NOW());
select
sum(l_extendedprice * l_discount)
as revenue
from
lineitem
where
l_shipdate >= date '1994-01-01'
and l_shipdate <  date '1994-01-01'
+interval '1' year
and l_discount between 0.09 - 0.01
and 0.09 + 0.01
and l_quantity < 24;
go
insert into TIMES values
('Q6 in Str02 end',NOW());

insert into TIMES values
('Q17 in Str02 start',NOW());
select
sum(l_extendedprice) / 7.0
as avg_yearly
from
lineitem,
part
where
p_partkey = l_partkey
and p_brand = 'Brand#53'
and p_container = 'JUMBO PKG'
and l_quantity < (
select
0.2 * avg(l_quantity)
from
lineitem
where
l_partkey = p_partkey
);
go
insert into TIMES values
('Q17 in Str02 end',NOW());

insert into TIMES values
('Q14 in Str02 start',NOW());
select
100.00 * sum(case
```

```
when p_type like 'PROMO%'
then l_extendedprice * (1 - l_discount)
else 0
end) / sum(l_extendedprice
* (1 - l_discount)) as promo_revenue
from
lineitem,
part
where
l_partkey = p_partkey
and l_shipdate >= date '1995-03-01'
and l_shipdate <  date '1995-03-01'
+interval '1' month
go
insert into TIMES values
('Q14 in Str02 end',NOW());

insert into TIMES values
('Q16 in Str02 start',NOW());
select
p_brand,
p_type,
p_size,
count(distinct ps_suppkey)
as supplier_cnt
from
partsupp,
part
where
p_partkey = ps_partkey
and p_brand <> 'Brand#30'
and p_type not like 'LARGE POLISHED%'
and p_size
in (7, 23, 19, 11, 10, 41, 48, 44)
and ps_suppkey not in (
select
s_suppkey
from
supplier
where
s_comment like '%Customer%Complaints%'
)
group by
p_brand,
p_type,
p_size
order by
supplier_cnt desc,
p_brand,
p_type,
p_size;
go
insert into TIMES values
('Q16 in Str02 end',NOW());

insert into TIMES values
('Q19 in Str02 start',NOW());
select
sum(l_extendedprice* (1 - l_discount))
as revenue
from
lineitem,
part
where
(
p_partkey = l_partkey
and p_brand = 'Brand#41'
and p_container
in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
and l_quantity >= 1 and l_quantity <= 1 + 10
and p_size between 1 and 5
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#21'
and p_container
in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
and l_quantity >= 19 and l_quantity <= 19 + 10
and p_size between 1 and 10
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#11'
and p_container
in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
and l_quantity >= 24 and l_quantity <= 24 + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON'
);
go
insert into TIMES values
('Q19 in Str02 end',NOW());

insert into TIMES values
('Q10 in Str02 start',NOW());
select
c_custkey,
c_name,
sum(l_extendedprice * (1 - l_discount))
as revenue,
c_acctbal,
n_name,
c_address,
c_phone,
c_comment
from
customer,
orders,
lineitem,
nation
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate >= date '1994-08-01'
and o_orderdate <  date '1994-08-01'
+interval '3' month
and l_returnflag = 'R'
and c_nationkey = n_nationkey
group by
c_custkey,
c_name,
c_acctbal,
c_phone,
n_name,
c_address,
c_comment
order by
revenue desc
limit 20;
go
insert into TIMES values
('Q10 in Str02 end',NOW());

insert into TIMES values
('Q9 in Str02 start',NOW());
select
nation,
o_year,
sum(amount) as sum_profit
from
(
select
n_name as nation,
extract(year from o_orderdate) as o_year,
l_extendedprice * (1 - l_discount)
- ps_supplycost * l_quantity as amount
from
part,
supplier,
lineitem,
partsupp,
orders,
nation
where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like '%blush%'
) as profit
group by
nation,
o_year
order by
nation,
```

```
o_year desc;
go
insert into TIMES values
('Q9 in Str02 end',NOW());

insert into TIMES values
('Q2 in Str02 start',NOW());
select
s_acctbal,
s_name,
n_name,
p_partkey,
p_mfgr,
s_address,
s_phone,
s_comment
from
part,
supplier,
partsupp,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and p_size = 3
and p_type like '%STEEL'
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'AFRICA'
and ps_supplycost = (
select
min(ps_supplycost)
from
partsupp,
supplier,
nation,
region
where
p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'AFRICA'
)
order by
s_acctbal desc,
n_name,
s_name,
p_partkey
limit 100;
go
insert into TIMES values
('Q2 in Str02 end',NOW());

insert into TIMES values
('Q15 in Str02 start',NOW());

select
s_suppkey,
s_name,
s_address,
s_phone,
total_revenue
from
supplier,
revenue2
where
s_suppkey = supplier_no
and total_revenue = (
select
max(total_revenue)
from
revenue2
)
order by
s_suppkey;

drop view revenue2;
go
insert into TIMES values
('Q15 in Str02 end',NOW());

insert into TIMES values
('Q8 in Str02 start',NOW());
select
o_year,
sum(case
when nation = 'FRANCE' then volume
else 0
end) / sum(volume) as mkt_share
from
(
select
extract(year from o_orderdate) as o_year,
l_extendedprice * (1 - l_discount) as volume,
n2.n_name as nation
from
part,
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2,
region
where
p_partkey = l_partkey
and s_suppkey = l_suppkey
and l_orderkey = o_orderkey
and o_custkey = c_custkey
and c_nationkey = n1.n_nationkey
and n1.n_regionkey = r_regionkey
and r_name = 'EUROPE'
and s_nationkey = n2.n_nationkey
and o_orderdate between
date '1995-01-01'
and date '1996-12-31'
and p_type = 'STANDARD POLISHED TIN'
) as all_nations
group by
o_year
order by
o_year;
go
insert into TIMES values
('Q8 in Str02 end',NOW());

insert into TIMES values
('Q5 in Str02 start',NOW());
select
n_name,
sum(l_extendedprice * (1 - l_discount))
as revenue
from
customer,
orders,
lineitem,
supplier,
nation,
region
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'EUROPE'
and o_orderdate >= date '1994-01-01'
and o_orderdate <  date '1994-01-01'
+interval '1' year
group by
n_name
order by
revenue desc;
go
insert into TIMES values
('Q5 in Str02 end',NOW());

insert into TIMES values
('Q22 in Str02 start',NOW());
select
cntrycode,
count(*) as numcust,
sum(c_acctbal) as totacctbal
from
(
select
substring(c_phone, 1, 2) as cntrycode,
c_acctbal
from
customer
where
```

```
substring(c_phone, 1, 2) in
('29', '14', '30', '28',
'31', '19', '33')
and c_acctbal > (
select
avg(c_acctbal)
from
customer
where
c_acctbal > 0.00
and substring(c_phone from 1 for 2) in
('29', '14', '30', '28',
'31', '19', '33')
)
and not exists (
select
*
from
orders
where
o_custkey = c_custkey
)
) as custsale
group by
cntrycode
order by
cntrycode;
go
insert into TIMES values
('Q22 in Str02 end',NOW());

insert into TIMES values
('Q12 in Str02 start',NOW());
select
l_shipmode,
sum(case
when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1
else 0
end) as high_line_count,
sum(case
when o_orderpriority <> '1-URGENT'
and o_orderpriority <> '2-HIGH'
then 1
else 0
end) as low_line_count
from
orders,
lineitem
where
o_orderkey = l_orderkey
and l_shipmode in ('REG AIR', 'MAIL')
and l_commitdate < l_receiptdate
and l_shipdate < l_commitdate
and l_receiptdate >= date '1995-01-01'
and l_receiptdate <  date '1995-01-01'
+interval '1' year
group by
l_shipmode
order by
l_shipmode;
go
insert into TIMES values
('Q12 in Str02 end',NOW());

insert into TIMES values
('Q7 in Str02 start',NOW());
select
supp_nation,
cust_nation,
l_year,
sum(volume) as revenue
from
(
select
n1.n_name as supp_nation,
n2.n_name as cust_nation,
extract(year from l_shipdate) as l_year,
l_extendedprice * (1 - l_discount) as volume
from
supplier,
lineitem,
orders,
customer,
nation n1,
nation n2
where
s_suppkey = l_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and (
(n1.n_name = 'UNITED STATES'
and n2.n_name = 'FRANCE')
or (n1.n_name = 'FRANCE'
and n2.n_name = 'UNITED STATES')
)
and l_shipdate between
date '1995-01-01'
and date '1996-12-31'
) as shipping
group by
supp_nation,
cust_nation,
l_year
order by
supp_nation,
cust_nation,
l_year;
go
insert into TIMES values
('Q7 in Str02 end',NOW());

insert into TIMES values
('Q13 in Str02 start',NOW());
select
c_count,
count(*) as custdist
from
(
select
c_custkey,
count(o_orderkey)
from
customer left outer join orders on
c_custkey = o_custkey
and o_comment not
like '%unusual%accounts%'
group by
c_custkey
) as c_orders (c_custkey, c_count)
group by
c_count
order by
custdist desc,
c_count desc;
go
insert into TIMES values
('Q13 in Str02 end',NOW());

insert into TIMES values
('Q18 in Str02 start',NOW());
select
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice,
sum(l_quantity)
from
customer,
orders,
lineitem
where
o_orderkey in (
select
l_orderkey
from
lineitem
group by
l_orderkey having
sum(l_quantity) > 314
)
and c_custkey = o_custkey
and o_orderkey = l_orderkey
group by
c_name,
c_custkey,
o_orderkey,
o_orderdate,
o_totalprice
order by
```

```
o_totalprice desc,
o_orderdate
limit 100;
go
insert into TIMES values
('Q18 in Str02 end',NOW());

insert into TIMES values
('Q1 in Str02 start',NOW());
select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount))
as sum_disc_price,
sum(l_extendedprice * (1 - l_discount)
* (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= date '1998-12-01'
-interval '82' day
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
go
insert into TIMES values
('Q1 in Str02 end',NOW());

insert into TIMES values
('Q4 in Str02 start',NOW());
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >= date '1993-05-01'
and o_orderdate < date '1993-05-01'
+interval '3' month
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
go
insert into TIMES values
('Q4 in Str02 end',NOW());

insert into TIMES values
('Q20 in Str02 start',NOW());
select
s_name,
s_address
from
supplier,
nation
where
s_suppkey in (
select
ps_suppkey
from
partsupp
where
ps_partkey in (
select
p_partkey
from
part
where
p_name like 'goldenrod%'
)
and ps_availqty > (
select
0.5 * sum(l_quantity)
from
lineitem
where
l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and l_shipdate >= date '1994-01-01'
and l_shipdate < date '1994-01-01'
+interval '1' year
)
)
and s_nationkey = n_nationkey
and n_name = 'IRAN'
order by
s_name;
go
insert into TIMES values
('Q20 in Str02 end',NOW());

insert into TIMES values
('Q3 in Str02 start',NOW());
select
l_orderkey,
sum(l_extendedprice * (1 - l_discount))
as revenue,
o_orderdate,
o_shippriority
from
customer,
orders,
lineitem
where
c_mktsegment = 'FURNITURE'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-01'
and l_shipdate > date '1995-03-01'
group by
l_orderkey,
o_orderdate,
o_shippriority
order by
revenue desc,
o_orderdate
limit 10;
go
insert into TIMES values
('Q3 in Str02 end',NOW());

insert into TIMES values
('Q11 in Str02 start',NOW());
select
ps_partkey,
sum(ps_supplycost * ps_availqty) as value
from
partsupp,
supplier,
nation
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'UNITED STATES'
group by
ps_partkey having
sum(ps_supplycost * ps_availqty) > (
select
sum(ps_supplycost * ps_availqty)
* 0.0001000000
from

supplier,
nation,
partsupp,
where
ps_suppkey = s_suppkey
and s_nationkey = n_nationkey
and n_name = 'UNITED STATES'
)
order by
value desc;
go
insert into TIMES values
('Q11 in Str02 end',NOW());
```

```
insert into TIMES values
('Q21 in Str02 start',NOW());
select
s_name,
count(*) as numwait
from
supplier,
lineitem l1,
orders,
nation
where
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (
select
*
from
lineitem l2
where
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (
select
*
from
lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
and n_name = 'INDONESIA'
group by
s_name
order by
numwait desc,
s_name
limit 100;
go
insert into TIMES values
('Q21 in Str02 end',NOW());

insert into TIMES values
('Str02 end',NOW());
go
```

B.4 Load Test

load_mysql.cmd

```
rem Load Test

set mysqlbin=C:\Program Files\MySQL\
MySQL Server 5.1\bin
set mysqldata=C:\Documents and Settings\
All Users\Application Data\MySQL\
MySQL Server 5.1\data
set cwd="%cd%"
cd %mysqlbin%

rem Creating Tables, constraints and indexes

mysql tpch < %cwd%\MySQL_Files\CreateTables.sql

rem Executing Bulk Inserts

move "%mysqldata%\temp1\*.*" "%mysqldata%\tpch"

mysql tpch < %cwd%\MySQL_Files\Inserts.sql
pause
move "%mysqldata%\tpch\*.tbl"
"%mysqldata%\temp1"
move "%mysqldata%\tpch\*.u1"
"%mysqldata%\temp1"
move "%mysqldata%\tpch\*.u2"
"%mysqldata%\temp1"
move "%mysqldata%\tpch\*.1"
"%mysqldata%\temp1"
move "%mysqldata%\tpch\*.2"
"%mysqldata%\temp1"

rem Creating Statistics

mysql tpch < %cwd%\MySQL_Files\
CreateStatistics.sql

rem Installing Refresh Functions

mysql tpch < %cwd%\MySQL_Files\CreateRF1.sql
mysql tpch < %cwd%\MySQL_Files\CreateRF2.sql

mysql tpch < %cwd%\MySQL_Files\LoadEnd.sql

cd %cwd%
semaphore -release SEM1

exit /B
```

B.5 Performance Test

run_mysql.cmd

```
rem Performance Test

set mysqlbin=C:\Program Files\MySQL\
MySQL Server 5.1\bin
set cwd="%cd%"
cd %mysqlbin%

rem Power Test

mysql < %cwd%\MySQL_Files\RF1Power.sql

mysql < %cwd%\MySQL_Files\Stream00.sql

mysql < %cwd%\MySQL_Files\RF2Power.sql

rem Throughput Test

cd %cwd%
start cmd /C RunStream01MySQL.cmd
start cmd /C RunStream02MySQL.cmd
semaphore -wait SEM2
semaphore -wait SEM2

cd %mysqlbin%

mysql < %cwd%\MySQL_Files\
RFsThroughput.sql

cd %cwd%
semaphore -release SEM1

exit /B
```

RunStream01MySQL.cmd

```
cd C:\Program Files\mysql\
mysql Server 5.1\bin

mysql < C:\MySQL_Files\
Stream01.sql

cd C:\

semaphore -release SEM2

exit /B
```

RunStream02MySQL.cmd

```
cd C:\Program Files\mysql\
mysql Server 5.1\bin

mysql < C:\MySQL_Files\
Stream02.sql
```

```
cd C:\

semaphore -release SEM2

exit /B
```

RF1Power.cmd

```
use tpch;

insert into TIMES values
('Power start',NOW());

insert into TIMES values
('Str00 RF1 start',NOW());

CALL RF1(40);

insert into TIMES values
('Str00 RF1 end',NOW());

insert into TIMES values
('Str00 start',NOW());
```

RF2Power.cmd

```
use tpch;

insert into TIMES values
('Str00 start',NOW());

insert into TIMES values
('Str00 RF2 start',NOW());

CALL RF2(32);

insert into TIMES values
('Str00 RF2 end',NOW());

insert into TIMES values
('Power end',NOW());

insert into TIMES values
('Throughput start',NOW());
```

RFsThroughput

```
use tpch;

insert into TIMES values
('Str01 RF1 start',NOW();

CALL RF1(4008);

insert into TIMES values
('Str01 RF1 end',NOW();

insert into TIMES values
('Str01 RF2 start',NOW();

CALL RF2(4000);

insert into TIMES values
('Str01 RF2 end',NOW();

insert into TIMES values
('Str02 RF1 start',NOW();

CALL RF1(7976);

insert into TIMES values
('Str02 RF1 end',NOW();

insert into TIMES values
('Str02 RF2 start',NOW();

CALL RF1(7968);

insert into TIMES values
('Str02 RF2 end',NOW();

insert into TIMES values
('Throughput end',NOW();
```

B.6 Full Test

all_tests_MySQL.cmd

```
start cmd /C load_mysql.cmd
semaphore -wait SEM1


cd C:\Program Files\mysql\
mysql Server 5.1\bin

mysql -u root -p -D tpch
--tee=MySQL_Results\load.txt -e
"select * from loadtimes
order by timestamp;"

cd C:\

start cmd /C run.cmd
semaphore -wait SEM1

cd C:\Program Files\mysql\
mysql Server 5.1\bin

mysql -u root -p -D tpch
--tee=MySQL_Results\performance.txt -e
"select * from times
order by start;"

exit /B
```

B.7 Concurrency Handling

semaphore.cpp

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <string.h>
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

int main(int argc, char **argv)
{
typedef enum {eUnknown, eStart,
eWait, eRelease} OPERATION;
OPERATION eOP = eUnknown;
int i;
HANDLE hSemaphore;

if (_stricmp(argv[1], "-wait") == 0)
eOP = eWait;
else if (_stricmp(argv[1], "-release") == 0)
eOP = eRelease;

if (eOP == eWait)
{
hSemaphore = CreateSemaphore(NULL, 0,
2000000000,argv[2]);
for (i=0; i<2; i++)
{
WaitForSingleObject(hSemaphore,
INFINITE);
}
CloseHandle(hSemaphore);
}

else if (eOP == eRelease)
{
hSemaphore = OpenSemaphore
(SEMAPHORE_MODIFY_STATE, FALSE, argv[2]);
ReleaseSemaphore(hSemaphore, 1, NULL);
CloseHandle(hSemaphore);
}

return 0;
}
```