Why You Should Run TPC-DS:A Workload Analysis

Meikel Poess
Oracle Corporation
400 Oracle Parkway
Redwood Shores, CA-94065, USA
650-633-8012
Meikel.Poess@oracle.com

Raghunath Othayoth Nambiar
Hewlett-Packard Company
20555 Tomball Parkway
Houston, TX-77070, USA
281-518-2748
Raghu.Nambiar@hp.com

David Walrath
Sybase Inc
561 Virginia Road
Concord, MA-01742, USA
978-287-1784
David.Walrath@sybase.com

ABSTRACT

The Transaction Processing Performance Council (TPC) is completing development of TPC-DS, a new generation industry standard decision support benchmark. The TPC-DS benchmark, first introduced in the "The Making of TPC-DS" [9] paper at the 32nd International Conference on Very Large Data Bases (VLDB), has now entered the TPC's "Formal Review" phase for new benchmarks; companies and researchers alike can now download the draft benchmark specification and tools for evaluation. The first paper [9] gave an overview of the TPC-DS data model, workload model, and execution rules. This paper details the characteristics of different phases of the workload, namely: database load, query workload and data maintenance; and also their impact to the benchmark's performance metric. As with prior TPC benchmarks, this workload will be widely used by vendors to demonstrate their capabilities to support complex decision support systems, by customers as a key factor in purchasing servers and software, and by the database community for research and development of optimization techniques.

1. INTRODUCTION

The TPC-DS benchmark, first introduced in the "The Making of TPC-DS" [9] paper at the 32nd International Conference on Very Large Data Bases (VLDB), has now entered the TPC's "Formal Review" phase for new benchmarks. During this step TPC member companies and researchers alike are encouraged to download the benchmark specification including the TPC provided tools, i.e. data generator and query generator, to evaluate the benchmark. As a consequence of this process the final version of TPC-DS might differ slightly from the version presented in this paper.

TPC-DS is intended to provide a fair and honest comparison of various vendor implementations by providing highly comparable, controlled and repeatable tasks in evaluating the performance of decision support systems (DSS). Its workload is expected to test the upward boundaries of hardware system performance in the areas of CPU utilization, memory utilization, I/O subsystem utilization and the ability of the operating system and database software to perform various complex functions important to DSS - examine large volumes of data, compute and execute the best execution plan for queries with a high degree of complexity, schedule efficiently a large number of user sessions, and give answers to critical business questions. In [9] we introduced the pivotal parts of TPC-DS, such as schema, data set, workload, metric and execution rules including the reasoning behind key decisions. In this paper we focus on a detailed performance analysis of TPC-DS' workload.

VLDB '07, September 23-28, 2007, Vienna, Austria.

TPC-DS' workload consists of three distinct disciplines: Database Load, Query Run and Data Maintenance. The query run is executed twice, once before and once after the data maintenance step. Each query run executes the same 99 query templates with different bind variables in permutated order, thereby simulating a workload of multiple concurrent users accessing the system. Using the terminology introduced in [9] these steps are executed in the following order:

Figure 1: TPC-DS Execution Order

Using an arithmetic mean the elapsed times for these three disciplines are combined into the primary performance metric, called QphH@SF, in the following way:

$$QphDS@SF = SF * 3600 \frac{198 * S}{(T_{QR1} + T_{DM} + T_{QR2} + 0.01 * S * T_{Load})}$$

T_{QR1}:elapsed time of Query Run 1
T_{QR2}:elapsed time of Query Run 2.
T_{DM}:elapsed time of the Data Maintenance run.
T_{Load}: elapsed time of database load test.
S:number of simulated concurrent users (streams)
SF:scale factor.

Figure 2: TPC-DS Primary Metric

The Performance Metric reflects the effective query throughput per second. The numerator represents the total number of queries executed on the system "198 * S", where 198 is the 99 individual queries times two query runs and S is the number of concurrent simulated users. The denominator represents the total elapsed time as the sum of Query Run1, Data Maintenance Run, Query Run 2 and a fraction of the Load Time. Note that the elapsed time of the data maintenance run is the aggregate of S executions of all data maintenance functions (see Section 5). By dividing the total number of queries by the total elapsed time, this metric represents queries executed per time period. Using an arithmetic mean to compute the primary benchmark metric should cause DBMS developers to concentrate primarily on long-running queries first, and then progressively continue with shorter queries. This generally matches the normal business case, where customers spend most of their tuning resources on the slowest queries.

The significant number of queries to optimize will also provide a test bed for self tuning databases since the diverse query set inflicts fluctuating resource pressures on the system. Oscillating

between CPU, IO (large and small) and Memory intensive queries, a system that can adapt to the resource needs of every query should excel in this type of workload.

Using existing query optimizers of three commercial DBMS products, Naveen Reddy and Jayant Haritsa analyzed their ability to generate the optimal plan of TPC-H query templates varying selectivity predicates [7]. Their results, reported at VLDB 2005, show that current query optimizers generate a variety of query plans, some of which are sub-optimal, when selectivity predicates are varied. The next generation query optimizers need to generate consistently good and yet simple query plans under any circumstances. TPC-DS with its 99 query templates and large schema is an excellent benchmark for testing query optimizers.

Analyzing a workload independently of a specific system is challenging. Apart from mainstream database management systems (DBMS), such as multi purpose relational databases systems (RDBMS) and specialized on line analytical processing systems (OLAPS) there are technologies emerging with new ideas on how to efficiently execute business intelligence queries on very large databases. Michael Stonebreaker's C-Store, presented at VLDB 2005, is just one example for new approaches to increasing query processing by utilizing innovative storage methodologies [6]. This paper focuses on analyzing TPC-DS' workload without any specific architecture in mind. The paper is divided in three large areas. The first part (Section 2) describes TPC-DS' key schema and data set characteristics, laying the foundation for the main part, the detailed analysis of the three disciplines of TPC-DS' workload: Database Load (Section 3), Query Run (Section 4) and Data Maintenance (Section 5). In the third part (Section 6) we conclude our paper by globally analyzing all three disciplines. Using different scenarios we show the impact of changes in the three workload disciplines on the primary metric.

2. KEY SCHEMA AND DATA SET CHARACTERISTICS

The design of the data set is motivated by the need to challenge the statistic gathering algorithms used for deciding the optimal query plan and the data placement algorithms, such as clustering and partitioning. TPC-DS uses a hybrid approach of data domain and data scaling [1][15]. While pure synthetic data generators have great advantages TPC-DS uses a mixture of both synthetic and real world based data domains. Synthetic data sets are well understood, easy to define and implement. However, following the TPC's paradigm to create benchmarks that businesses can relate to, a hybrid approach to data set design scores many advantages over both pure synthetic and pure real world data. This approach allows both realistically skewed data distributions yet still a predictable workload.

Compared to previous TPC decision support benchmarks [12] TPC-DS uses much wider tables, of up to 39 columns with domains ranging from integer, float (with various precisions), char, varchar (of various lengths) and date. Combined with a large number of tables (total of 25 tables and 429 columns) the schema gives both the opportunity of a realistic and challenging query set (see Section 4) as well as an opportunity for innovative data placement algorithms and other schema optimizations, such as complex auxiliary data structures. The number of times columns are referenced in the dataset varies between 0 and 189. Figure 3 shows the number of column references, classified in buckets, for those columns that are referenced. Of those columns accessed, the largest number of columns are referenced between 5 and 49 times. The large column set and diverse query set of TPC-DS also protects its metric from unrealistic tuning and artificial inflations of the metric, a problem which rapidly destroyed the usefulness of an older decision support benchmark, TPC-D, in the late 1990s. That, combined with the complex data maintenance functions (see Section 5) and load time participating in the primary performance metric creates the need for fast and efficient algorithms to create and maintain auxiliary data structures and the invention of new algorithms.

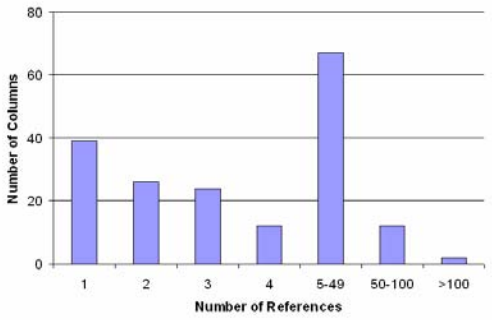

Figure 3: Number of Column References

The introduction of NULL values into any column except the primary keys opens yet another dimension of challenges for the query optimizer compared to prior TPC decision support benchmarks. The percent of NULL values in each non-primary key column varies from 4 to 100 percent based on the column. Most columns have 4 percent NULL values. The important `rec_end_date` columns (see 5.1) have 50 percent NULL values. Some columns were unused (total of 236) or intentionally left entirely NULL for future enhancements of the query set.

3. DATABASE LOAD

The process of creating a TPC-DS test database is denoted as the database load. It consists of both hardware and database preparation steps, some of which are timed and un-timed. All steps must be fully disclosed in the benchmark publication, called the full disclosure report, as per TPC guidelines. The exact steps executed in a benchmark publication depend on the specific system and database implementation. However, the following table lists common database load steps.

Database Load Step	Timed
System preparation	no
Flat file generation	no
Permutation of flat file rows	no
Database creation	no
Tablespace creation	no
Loading of base tables	yes
Creation and validation of constraints.	yes
Creation of auxiliary data structures	yes
Analysis of tables and auxiliary data structures	yes

Table 1: Database Load Steps

In order to unambiguously measure the timed steps of the database load on different DBMS implementations, the benchmark defines the start of the load either immediately after the tables and auxiliary data structures are created or when the first character of any flat file is accessed, whichever occurs first. It ends when the database is fully configured for the performance test.

The database can be loaded from flat files or in-line using,

for example, named pipes. If data is loaded from flat files, the time to generate the data is not timed, but the storage to host the files must be priced. However, if an in-line method is used, the data is loaded as it is generated and, therefore, the generation is timed. The TPC-DS data generator (dbgen2) can be fully parallelized. To fully utilize one CPU, a storage subsystem must support about 400 IOs per second. Assuming an adequate IO subsystem, it takes about 20 seconds to generate 1GByte of data using one core of a current x86-class processor. So on a four core x86-class processor system, a 100Gbyte database can be generated in about 500s seconds. As noted above, if data is loaded in-line, this overhead will be included in the total elapsed time of the load. [2] describes dbgen2 and its internals more in detail.

But why test the database load at all? Isn't a database loaded once and then only incrementally maintained? It turns out that there are many reasons DSS get reloaded in real life. Most DSS outgrow capacity after a few years of operation demanding hardware upgrades, which result in reloading the database in most cases. Another reason for reload is a significant change in data distributions. For instance, if the data distribution in the partitioning column changes dramatically so that data is no longer evenly distributed across disks, a reload might be necessary to allow even usage of the entire system.

There is also an important benchmark purpose for measuring the initial database load and making it part of the primary benchmark metric. Having the load as a timed portion in the overall metric is a very important step in allowing complex auxiliary structures commonly used in DSS, yet, having a robust workload that cannot be easily broken with clever auxiliary structures as happened to TPC-D. Table 1 lists the measured steps of the load. Dbgen2 generates flat files that correspond in structure to those of the data warehouse tables. That is, no complex restructuring of data is required (see Section 5). The first step, Loading of Base Tables, measures how fast a system can read the input data and convert the text into its internal binary representation (a.k.a. database pages). The next two steps measure how fast constraints and auxiliary data structures such as indexes and materialized views are created and validated. The last step measures how fast the system analyses the data that has been loaded. This is particularly important since the TPC-DS data is highly skewed (see [2] and [9] for more details).

4. QUERY RUN

In order to address the enormous range of query types and user behaviors encountered by decision support systems, TPC-DS utilizes a generalized query model. This model utilizes 99 query templates, which enable the data generator (qgen2) to generate virtually any number of SQL queries by means of query substitution [2]. The query templates test the interactive and iterative nature of on-line analytical processing (OLAP) queries, the extraction of queries for data mining tools, ad-hoc queries, and the more planned behavior of well known reporting queries.

A query run is clearly central to the TPC-DS benchmark. It is executed twice and both runs are counted in the primary metric, while the load and data maintenance are each executed once. Each query run executes 20 or more concurrent query streams. The multiple query streams simulate many users executing queries against the database concurrently. (see [9] for more details).

Running multiple streams concurrently on a system imposes challenges both on the system's hardware and software. In order to accommodate multiple streams the hardware needs to be sized such that each user has enough memory to execute resource intensive operations such as joins and sorts. It also needs to be sized so that all users have enough temporary space to execute queries with large intermediate results sets. The DBMS also needs to find the optimal execution plan for the multiple concurrent users. An optimal execution plan, including the execution algorithms and degree of parallelism, depend on both the query being executed and the available system resources at time of execution. In an extreme case serial execution might be optimal: when a system is heavily loaded due to other work, running with less parallelism prevents the system from being overloaded; overall system throughput often can benefit, and even the query being optimized sometimes can benefit from executing a query using a serial execution plan because of the overhead of parallel execution. In case of a Grid system (multiple nodes) smart parallelization, e.g. allocation of work units across query execution nodes is essential for optimal performance [3]. It is also essential that the DBMS and the operating system (OS) are orchestrated to work together. In commercial DSS products without active resource management, the OS is in charge of scheduling resources such as memory and IO. It is becoming more and more important that the DBMS takes on this responsibility. In general a frugal system that economically assigns resources and manages resources dynamically will excel in a multi stream run.

Nevertheless, in order to fully understand the query workload it is essential to understand both each query and the query mix. In the following sections we analyze all 99 query templates from different perspectives. First, we characterize them by the query classes that TPC-DS defines, followed by a characterization by hardware resources and concluded with a characterization by SQL feature. In each section we identify queries that are discussed in detail at the end of Section 4.

4.1 Characterization by Query Class

A sophisticated decision support system must support a diverse user population. While there are many ways to categorize those diverse users and the queries that they generate, TPC-DS has defined four broad non mutually exclusive query classes that, taken together, characterize most decision support queries: Reporting, Ad Hoc, Iterative and Data Mining (Table 2).

Query Class	Number of Queries
Reporting Class	41
Ad Hoc Class	59
Iterative Class	4
Data Mining Class	23

Table 2: Characterization by Query Class

Reporting queries capture the "reporting" portion of a DSS system. They include queries that are executed periodically to answer well-known, pre-defined questions about the financial and operational health of a business. Although reporting queries tend to be static, minor changes are common. From one use of a given reporting query to the next, a user might choose to shift focus by varying a date range, geographic location or a brand name. TPC-DS defines 41 reporting query templates.

Ad hoc queries capture the dynamic nature of a DSS system, in which impromptu queries are constructed to answer immediate and specific business questions. The central difference between ad hoc queries and reporting queries is the limited degree of foreknowledge that is available to the DBA when planning for an ad hoc query. Other than the most generic schema optimizations,

with ad-hoc queries a DBA has little or no foreknowledge of useful physical data layout (e.g. clustering and partitioning) or optimal auxiliary data structures (e.g. indexes and materialized views). TPC-DS defines 59 ad hoc query templates.

Amalgamating both types of queries has been traditionally difficult in benchmark environments since per the definition of the benchmark all queries, apart from bind variables, are known in advance. TPC-DS accomplishes this fusion by dividing the schema into reporting and ad hoc "parts" by a ratio of 4/6. . The store and web sales channels constitute an ad hoc portion of the schema, while the catalog sales channel constitutes the reporting part. For the catalog sales channel complex auxiliary data structures are allowed, while for the other two channels only simple, basic auxiliary data structures are allowed. Hence, queries predominantly accessing the ad hoc part constitute the ad hoc query set while the queries predominantly accessing the reporting part are considered the reporting query templates.

Iterative queries allow for the exploration and the analysis of business data to discover new and meaningful relationships and trends. While this class of queries is similar to the Ad Hoc Query class, it is distinguished by a scenario-based user session, in which a sequence of queries is submitted, with one leading to another, and where the sequence may include both complex and simple queries. There are 4 iterative query templates defined.

Data mining is the process of sifting through large amounts of data to produce data content relationships. It can predict future trends, allowing businesses to make proactive, knowledge-driven decisions. This class of queries typically consists of joins and large aggregations that return large data result sets (more than 1,000 rows) for extraction and further analysis in specialized data mining tools. TPC-DS defines 23 data mining query templates.

4.2 Characterization by Schema Coverage

The schema design is the foundation for a good query set. If the schema does not allow for the designing of queries that test the performance of all aspects of a DSS, the benchmark has not succeeded in a central goal. TPC-DS is designed with a multiple-snowflake schema allowing the exercise of all aspects of commercial DSS, built with a modern DBMS. The snowflake schema is designed using a retail model consisting of three sales channels plus an inventory fact table. Each sales channel consists of two fact tables each: sales and returns, while the inventory consists of one fact table. The corresponding sales and returns tables can be joined with their foreign key-primary key relationships. The fact tables of different sales channels can be joined using their shared dimensions, e.g. item or customer. For a detailed ER diagram please refer to [9] and [13].

This design allows for a rich query set. It allows query execution of both star schema and 3rd normal form (3NF) execution paths. Typical executions in a star schema might involve bitmap accesses, bitmap merges, bitmap joins and conventional index driven join operations. The access paths in a 3NF DSS system are often dominated by large hash or sort-merge joins, and conventional index driven joins are also common. In both systems large aggregations, which often include large sort operations are widespread. This diversity imposes challenges both on hardware and software systems. High sequential I/O-throughput is critical to excel in large hash-join operations. At the same time, index driven queries stress the I/O subsystem's ability to perform small random I/Os. Additionally, this diversity also challenges the query optimizer in its decision to either use a star schema approach, such as star transformation, or a more traditional approach, such as nested loops, hash-joins etc. This seems to be an area in which current query optimizers often have huge deficits. As explained in the previous section, the partitioning of the schema into three sales channels allow for amalgamating both ad hoc and reporting queries into the same benchmark. Thirdly, the large number of the tables allow for both a richness variety of query structures. It allows for queries accessing either a few or many tables, both fact and dimension tables as well as the creation of "hot" tables that are accessed in almost all queries.

	Schema Coverage	Number of Queries
Section 1	One Fact Table	54
	Multiple Fact Tables	39
	Only Dimension Tables	6
	Only Store Sales Channel	37
	Only Catalog Sales Channel	12
	Only Web Sales Channel	12
Section 2	date_dim	91
	store	68
	store_sales	64
	item	58
	customer	57
	catalog_sales	38
	web_sales	36
	customer_address	34
	customer_demographics	20
	household_demographics	17
	store_returns	15
	catalog_returns	12
	promotion	10
	warehouse	10
	web_returns	10
	call_center	6
	income_band	6
	time_dim	5
	web_site	5

Table 3: Query Characterization by Schema Coverage

Table 3 displays the schema coverage of all TPC-DS queries. The first section lists the number of queries that access interesting table combination. There are 54 individual queries that only access one fact table. These queries access either only one sales table, one return table or the inventory table, representing traditional star. A good example of this type of query is Query 81 (see Figure 8). There are also 39 queries that access more than one fact table. Queries of this type can further be divided into queries that join fact tables and those that union the results of fact tables. There are 17 queries that union sub-queries accessing fact tables and 22 queries that join fact tables either directly or thought SQL intersect or except operations. The later query type constitutes the more traditional execution of queries in a 3rd NF schema. They join large tables, thereby exercising large hash, merge, or index based joins, and in some cases produce large intermediate result sets and large sorts. A good example is Query 78 (see Figure 10).

Section 2 in Table 3 summarizes the table coverage of all queries. It shows that the Date_dim table is accessed in almost every table. This is not surprising because almost all queries group or constrain their result on days, weeks, months or years. It also shows that Store, Item, Customer are the most accessed dimensions (over 50% of the queries). The most accessed fact table is Store_sales with 64 references, followed by Catalog_sales and Web_sales with 38 and 36 references.

4.3 Characterization by Resource Utilization

Each decision support query has its own hardware resource utilization pattern, unique to the way it is executed on a particular system. On an SMP system the system resources that are considered most important, especially when sizing a system for a particular workload, are CPU, reads and writes from/to the disk subsystem, inter-node communication network and memory. In order to understand a workload it is essential to examine queries that exhibit extreme behaviors, such as CPU intensive queries or IO intensive queries, and that, at the same time, exhibit a simple structure. In TPC-H the most CPU intensive query is typically Query 1, while the most IO intensive query is typically Query 6. In this section we characterize TPC-DS query templates according to their IO and CPU consumption.

Figure 4 graphs the CPU and IO utilization of all 99 query templates. From each query template we generate one query using the default substitution parameters for a total of 99 SQL queries. We then run all queries sequentially on a commercial RDBMS and analyze their execution patterns for CPU utilization, and average MBytes per second of reads and writes which is combined into average IO utilization.

Due to decision support queries being frequently complex, often joining multiple tables requiring different join methods, sorting large amounts of data and computing aggregate data, their execution pattern is typically not in a steady state for a long time. E.g., the usual execution pattern of a hash join can be described as a relatively short burst of intensive IO during the creation of the hash table of the left side of the join followed by a longer, usually CPU bound phase where the right side of the join is scanned and probed into the hash table. Hence, one cannot infer any specific CPU/IO pattern from the two parameters above. However, they give a general idea how resource intensive queries are.

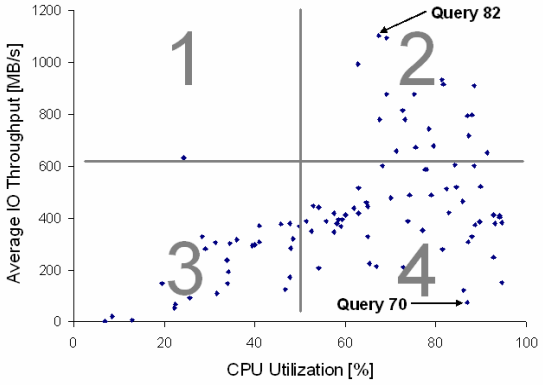

Figure 4: Characterization by Resource Utilization

Figure 4 graphs the average CPU and IO utilization in a two dimensional chart. The x-axis shows the average CPU utilization while the y-axis shows the average IO utilization. Each data point indicates one query. Dividing the chart into four quadrants we group queries with specific IO and CPU patterns. IO intensive queries with low CPU utilization are summarized in Quadrant 1 while CPU intensive queries with low IO utilization are grouped in Quadrant 3. Quadrant 2 groups IO and CPU intensive queries while Quadrant 4 groups high CPU and low IO intensive queries. Most queries are located in the fourth quadrants while quadrants 2 and 3 are equally populated. In Section 4.5 we analyze examples of pure CPU and pure IO intensive queries. Query 82 servers as an IO intensive example while Query 70 serves as a CPU intensive example.

4.4 Characterization by SQL Features

Database functionality has increased dramatically since the TPC introduced its first decision support benchmark. SQL functionality has especially increased and, thanks to ANSI, the syntax to express more complex queries concisely has been standardized. TPC-DS queries are mostly phrased in compliance with SQL1999 core with OLAP amendment. Table 4 shows how many TPC-DS queries make use of certain SQL constructs.

SQL Feature	Number of Queries
Common Sub-expression	31
Correlated Sub-Query	15
Uncorrelated Sub-Query	76
Group By	78
Order By	64
Rollup	9
Partition	11
Exists	5
Union	17
Intersect	2
Minus	1
Case	24
Having	5

Table 4: Characterization by SQL Functionality

4.5 Performance Analysis of Selected Queries

Figure 5 shows an IO intensive query, Query 82. It lists items that are offered from specific manufacturers, together with their current prices that were sold through the store sales channel. Only items in a given $30 price range that consistently had a quantity on hand in any warehouse between 100 and 500 in a 60-day period are chosen by this query. According to the classification in Section 4.1 Query 82 is an ad hoc query.

```
SELECT i_item_id
      ,i_item_desc
      ,i_current_price
FROM item, inventory
    ,date_dim ,store_sales
WHERE i_current_price between [P] and [P] + 30
  AND inv_item_sk = i_item_sk
  AND d_date_sk=inv_date_sk
  AND d_date between cast('[DATE]' as date)
                 AND (cast('[DATE]' as date)+60)
  AND i_manufact_id IN ([ID.1],[ID.2],[ID.3])
  AND inv_quantity_on_hand between 100 and 500
  AND ss_item_sk = i_item_sk
GROUP BY i_item_id
        ,i_item_desc
        ,i_current_price
ORDER BY i_item_id;
```

Figure 5: IO Intensive Query (Query 82)

Query 82 joins the dimensions Date_dim and Item with the two largest tables, Store_sales and Inventory causing IO intensive scans of vast amount of data. The simple local predicates "`i_manufact_id IN ([ID.1],[ID.2],[ID.3])`" and "`i_current_price between [P] and [P] + 30`" reduce the number of rows to be returned from the Item table to 0.15 percent and the simple local predicate "`d_date between cast('[DATE]' as date) AND(cast('[DATE]' as date)+60)`" reduce the number of rows to be return from the Date_dim table to 0.8 percent. Yet, large amounts of data need to

be scanned as no additional auxiliary data structures, such as indexes, are allowed on the inventory table, e.g. an index on the inv_quantity_on_hand column.In this query, the number of rows that are fed into the subsequent grouping and sort operations are so small that these operations, which tend to be more CPU intensive, contribute insignificantly to the overall query execution time. This query is classified as an ad hoc query.

```
SELECT
  sum(ss_net_profit) as total_sum
 ,s_state
 ,s_county
 ,grouping(s_state)+grouping(s_county)
 ,rank()over(partition by grouping(s_state)
                         +grouping(s_county)
            ,case when grouping(s_county)=0
             then s_state end
             order by sum(ss_net_profit) desc)
FROM store_sales
    ,date_dim
    ,store
WHERE d_year = [YEAR]
  AND d_date_sk = ss_sold_date_sk
  AND s_store_sk  = ss_store_sk
  AND s_state in
     (SELECT s_state
      FROM (SELECT
             s_state
            ,rank()over(partition by s_state
             order by sum(ss_net_profit)desc) as r
            FROM store_sales
                ,store
                ,date_dim
            WHERE d_year =[YEAR]
              AND d_date_sk = ss_sold_date_sk
              AND s_store_sk  = ss_store_sk
            GROU BY s_state)
       WHERE r <= 5)
 GROUP BY ROLLUP(s_state,s_county)
 ORDER BY
   lochierarchy desc
  ,CASE WHEN lochierarchy = 0 THEN s_state END
  ,rank_within_parent;
```

Figure 6: CPU Intensive Query (Query 70)

Query 70 in Figure 6 shows a CPU intensive query. It computes the store sales net profit ranking by state and county for a given year and determines the five most profitable states. This query joins the Store and Date_dim dimension tables with the store_sales table. The only projection predicate is the constraint on date "d_year = [YEAR]" causing large amount of data to be scanned. However, complex operations, such as the grouping/ranking functions, the aggregate functions and the case statements contribute to a high CPU overhead making this a very CPU intensive query. The fact that only the sales table is accessed classifies this query as an ad hoc query.

Figure 7 shows Query 40. It calculates the impact of an item price change on catalog sales by computing the total sales for items in a 30 day period before and after the price change. The items are grouped by the warehouse location that delivered them. This query is a reporting query because it accesses the catalog sales table and, therefore, complex auxiliary structures such as materialized views or bitmap indexes are allowed in order to optimize the execution of this query.

```
SELECT
   w_state
  ,i_item_id
  ,SUM(CASE WHEN d_date < '2000-03-11'
            THEN cs_sales_price-cr_refunded_cash
            ELSE 0 END) as sales_before
  ,SUM(CASE WHEN d_date >= '2000-03-11' as date
            THEN cs_sales_price-cr_refunded_cash
            ELSE 0 END) as sales_after
   FROM
   catalog_sales left outer join catalog_return
                      on(cs_item_sk = cr_item_sk
   and cs_order_number = cr_order_number)
  ,warehouse, item, date_dim
WHERE i_current_price BETWEEN 0.99 and 1.49
  AND i_item_sk = cs_item_sk
  AND cs_warehouse_sk = w_warehouse_sk
  AND cs_sold_date_sk = d_date_sk
  AND d_date between(cast('2000-03-11'as date)-30)
                 and(cast('2000-03-11'as date)+30)
 GROUP BY w_state,i_item_id
 ORDER BY w_state,i_item_id;
```

Figure 7: Reporting Query (Query 40)

```
WITH customer_total_return AS
 (select cr_returning_customer_sk as ctr_cust_sk
       ,ca_state as ctr_state
       ,sum(cr_return_amt_inc_tax) as ctr_return
  FROM catalog_returns
      ,date_dim
      ,customer_address
 WHERE cr_returned_date_sk = d_date_sk
   AND d_year =[YEAR]
   AND cr_returning_addr_sk = ca_address_sk
 GROUP BY cr_returning_customer_sk
        ,ca_state)
 SELECT c_customer_id,c_salutation
       ,c_first_name,c_last_name
       ,ca_street_number,ca_street_name
       ,ca_street_type,ca_suite_number
       ,ca_city,ca_county
       ,ca_state,ca_zip,ca_country,ca_gmt_offset
       ,ca_location_type,ctr_return
 FROM customer_total_return ctr1
     ,customer_address
     ,customer
 WHERE ctr1.ctr_return
       > (SELECT avg(ctr_return)*1.2
          FROM customer_total_return ctr2
          WHERE ctr1.ctr_state = ctr2.ctr_state)
          AND ca_address_sk = c_current_addr_sk
          AND ca_state = '[STATE]'
          AND ctr1.ctr_cust_sk = c_customer_sk
 ORDER BY c_customer_id,c_salutation,c_first_name
      ,c_last_name,ca_street_number,ca_street_name
      ,ca_street_type,ca_suite_number,ca_city
      ,ca_county,ca_state,ca_zip,ca_country
      ,ca_gmt_offset ,ca_location_type
      ,ctr_return;
```

Figure 8: Reporting Query/One Fact Table Access (Query 81)

Figure 8 is an example of a reporting query and one that accesses only one fact table. For a specific year and state it locates customers with bad item returning habits. It lists those customer names with their detailed contact information who have returned items that they had bought from the catalog more than 20 percent the average time their peer customers have returned items in the same time period. The selectivity predicate on year restricts the total data amount to 20% of the catalog return table. The state column in the Customer_address table is skewed, e.g. there are three groups of states: small, medium and large. The selectivity predicate on state selects a large state, hence selecting about 5 percent

of the dataset. Query 81 qualifies as a reporting query because it accesses the catalog sales channel. Data to speed up this query can be materialized or complex indexes can be defined.

```
WITH frequent_ss_items as
(SELECT substr(i_item_desc,1,30) itemdesc
       ,i_item_sk item_sk
       ,d_date solddate
       ,count(*) cnt
 FROM store_sales ,date_dim ,item
 WHERE ss_sold_date_sk = d_date_sk
   AND ss_item_sk = i_item_sk
   AND d_year between [YEAR] and [YEAR]+2
 GROUP BY substr(i_item_desc,1,30)
         ,i_item_sk
         ,d_date
 HAVING count(*) >4),
 max_store_sales AS
(SELECT MAX(csales) tpcds_cmax
  from
    (select c_customer_sk
           ,SUM(ss_quantity*ss_sales_price)
csales
     FROM store_sales ,customer ,date_dim
     WHERE ss_customer_sk = c_customer_sk
       AND ss_sold_date_sk = d_date_sk
       AND d_year between [YEAR] and [YEAR]+2
     GROUP BY c_customer_sk) x),
 best_ss_customer as
(SELECT c_customer_sk
       ,sum(ss_quantity*ss_sales_price) ssales
 FROM store_sales
     ,customer
 WHERE ss_customer_sk = c_customer_sk
 GROUP BY c_customer_sk
 HAVING sum(ss_quantity*ss_sales_price)
       >0.95*(SELECT *
               FROM max_store_sales))
 SELECY sum(sales)
 FROM (
   (SELECT cs_quantity*cs_list_price sales
    FROM catalog_sales ,date_dim
    WHERE d_year = [YEAR]
    AND d_moy = 7
    AND cs_sold_date_sk = d_date_sk
    AND cs_item_sk in (SELECT item_sk
                       FROM frequent_ss_items)
    AND cs_bill_customer_sk IN
                      (SELECT c_customer_sk
                       FROM best_ss_customer)
   )UNION ALL
   (SELECT ws_quantity*ws_list_price sales
    FROM web_sales ,date_dim
    WHERE d_year = [YEAR]
      AND d_moy = 7
      AND ws_sold_date_sk = d_date_sk
      AND ws_item_sk IN(select item_sk
                        FROM frequent_ss_items)
      AND ws_bill_customer_sk IN
                       (SELECT c_customer_sk
                        FROM best_ss_customer)));
```

Figure 9: Iterative Query (Query 24)

Figure 9 shows an example of an iterative query. Because of space constraints we print only the first iteration in its entirety. The following iterations of this query differ only in the predicates, which we list at the end of the query. The first iteration identifies customers whose sales amount is greater than 95% of the maximum customer sales in a 3-year period including those items they bought most frequently. For the purpose of this query the "most frequently sold items" are defined as items that were sold consistently more than 4 times a day in stores during the same 3 year period. The query returns the sales sum of these frequently sold items bought by the 5 percent of the best customers through the web and catalog sales channels in a specific month (July) of the last year in the above period.

Each of the following iterations in this query define the same common sub-expressions as the first iteration: frequent_ss_items, max_store_sales and best_ss_customer. The second iteration finds detailed information about the top customers. That is, instead of just returning the sum of sales, it returns the last name, first name and sales of the particular customer. The idea behind this iteration is that the user is first interested in the total sales and then drills down into who actually contributed to the total sales.

The third through the last iteration simulate the user drilling further into the Customer and Customer_demographics dimensions by adding additional predicates on the Customer and Customer_demographics dimensions:

Iteration 3: non-gift sales,
bill_customer = ship_to_customer

Iteration 4: customers with a specific gender,
cd_gender='m'

Iteration 5: customers with a specific marital status,
cd_marital_status='d'

Iteration 6: customers with a specific purchase estimate,
cd_purchase_estimate=10000

The following Query 64 in Figure 10 shows an example of a query joining multiple fact tables. The query is searching for patterns of returns between two years in certain price ranges for customers with certain demographic characteristics.

This query is interesting to the benchmark in a couple of aspects: it joins across 4 fact tables in two subject areas (store sales and web sales) within a common table expression, with a total of 19 table references in the common table expression's FROM clause. That common table expression is joined to itself in the outermost SELECT for a total of 38 table references in the query. By joining across fact tables, it creates potentially large many-to-many joins between the fact tables, which in combination with the number of tables in the FROM clauses gives a modestly complex join topology to optimize, where some joins may be large and difficult to estimate the output row counts.

```
WITH cross_sales AS
(select i_product_name product_name
  ,i_item_sk item_sk,w_state warehouse_state
  ,w_warehouse_name warehouse_name
  ,ad1.ca_street_number b_street_number
  ,ad1.ca_street_name b_streen_name
  ,ad1.ca_city b_city,ad1.ca_zip b_zip
  ,ad2.ca_street_number c_street_number
  ,ad2.ca_street_name c_street_name
  ,ad2.ca_city c_city
  ,ad2.ca_zip c_zip,d1.d_year as syear
  ,d2.d_year as fsyear,d3.d_year s2year
  ,count(*) cnt,sum(ss_wholesale_cost) s1
,sum(ss_sales_price) s2
    ,sum(ss_net_profit) s3
    ,sum(cs_wholesale_cost) s4
    ,sum(cs_sales_price) s5,sum(cs_net_profit) s6
    ,sum(ws_wholesale_cost) s7
    ,sum(ws_sales_price) s8,sum(ws_net_profit) s9
 FROM store_sales,store_returns
     ,web_sales,web_returns
     ,catalog_sales,catalog_returns
     ,date_dim d1,date_dim d2,date_dim d3
```

```
     ,warehouse,item,customer
     ,customer_demographics cd1
     ,customer_demographics cd2,promotion
     ,household_demographics hd1
     ,household_demographics hd2
     ,customer_address ad1,customer_address ad2
     ,income_band ib1 ,income_band ib2
 WHERE ws_warehouse_sk = w_warehouse_sk
  AND ss_sold_date_sk = d1.d_date_sk
  AND ws_bill_customer_sk = c_customer_sk
  AND ws_bill_cdemo_sk= cd1.cd_demo_sk
  AND ws_bill_hdemo_sk = hd1.hd_demo_sk
  AND ws_bill_addr_sk = ad1.ca_address_sk
  AND ss_item_sk = i_item_sk
  AND ss_item_sk = sr_item_sk
  AND ss_ticket_number = sr_ticket_number
  AND ss_item_sk = ws_item_sk
  AND ws_item_sk = wr_item_sk
  AND ws_order_number = wr_order_number
  AND ss_item_sk = cs_item_sk
  AND ws_item_sk = cr_item_sk
  AND cs_order_number = cr_order_number
  AND c_current_cdemo_sk = cd2.cd_demo_sk
  AND c_current_hdemo_sk = hd2.hd_demo_sk
  AND c_current_addr_sk = ad2.ca_address_sk
  AND c_first_sales_date_sk = d2.d_date_sk
  AND c_first_shipto_date_sk = d3.d_date_sk
  AND ws_promo_sk = p_promo_sk AND i_size='[SIZE]'
  AND hd1.hd_income_band_sk=ib1.ib_income_band_sk
  AND hd2.hd_income_band_sk=ib2.ib_income_band_sk
  AND cd1.cd_marital_status<>cd2.cd_marital_status
  AND ib1.ib_upper_bound between [INC1]
                             and [INC1] + 20000
  AND ib2.ib_upper_bound between [INC2] AND
                                 [INC2] + 20000
  AND hd1.hd_buy_potential = '1001-5000' and
  AND hd2.hd_buy_potential = '501-1000'
  AND (i_current_price between [P1] and [P1]+10
    OR i_current_price between [P2] and [P2]+10)
GROUP BY i_product_name,i_item_sk
        ,w_warehouse_name,w_state
        ,ad1.ca_street_number,ad1.ca_street_name
        ,ad1.ca_city,ad1.ca_zip
        ,ad2.ca_street_number,ad2.ca_street_name
        ,ad2.ca_city,ad2.ca_zip
        ,d1.d_year,d2.d_year,d3.d_year)
SELECT * FROM cross_sales cs1,cross_sales cs2
WHERE cs1.item_sk=cs2.item_sk
  AND cs1.syear = [YEAR] and cs2.syear = [YEAR] +1
ORDER BY cs1.product_name,cs1.warehouse_name
        ,cs2.cnt;
```

Figure 10: Query Joining Multiple Fact Tables (Query 64)

5. DATA MAINTENANCE

An important component in the life-cycle of a DSS is the maintenance of fact tables and slowly changing dimensions, commonly referred to as the ETL process (Extraction, Transformation and Load). Its performance is becoming more and more important as the time between database updates shrinks, especially for global 24x7 operations utilizing DSS. In extreme cases the time window for incremental maintenance is approaching zero making so-called trickle updates to database tables necessary. At the same time the number and complexity of auxiliary data structures that are commonly used in DSS to reduce query elapsed time are drastically increasing the execution time of the data maintenance processes.

Having realized that a successful benchmark, must have a narrow scope of components whose performance will bemeasured, TPC-DS defines a server centric data maintenance process, also referred to as ELT (Extraction, Load, Transformation). Contrary to traditional approaches that utilize specialized tools or custom written code, in TPC-DS only some form of SQL is allowed to implement the data maintenance process. Existing DBMS products in recent years have improved speed and functionality to the level where increasing numbers of customers are executing simple to moderately complex transformation processes in their warehouse DBMS rather than using a separate specialized tool. Advantages of ELT are:

- ELT is parallelized according to the data set, and disk I/O is usually optimized at the engine level for faster throughput,
- ELT scales with the existing hardware and RDBMS engine,
- even ignoring scalability, ELT leverages the existing DBMS and hardware for better ROI,
- ELT keeps all data in the DBMS all the time.

In TPC-DS data from operational systems is provided in the form of flat files, also referred to as the refresh data set. Each flat file of the refresh data set models the content of one table in the fictitious operational system. Taken together these tables constitute the schema of this system. Starting with the reading of the refresh data set, the ELT process in TPC-DS includes the integration and consolidation of data from operational systems, thereby applying diverse workload consisting of data transformations that range from simple string manipulations to complex 3rd normal form denormalizations and various algorithms to maintain slowly changing dimensions and fact tables. An overview of data maintenance is depicted in the following Figure 11.

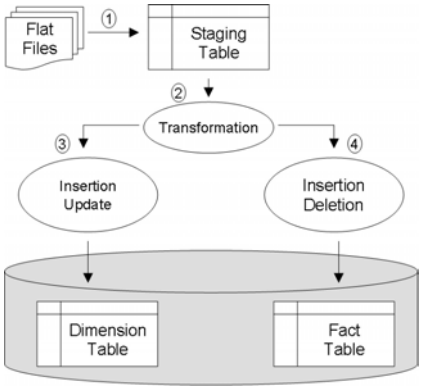

Figure 11: Data Maintenance Flow

Step 1 loads the refresh data set into internal tables. Each file is loaded into one table. Step 2 transforms internal tables so that the data can be loaded into the data warehouse tables. If additional disk space is necessary for this operation, a staging area can be used, which must be priced. There are several types of transformations as follows. Direct source to target transformation: This most common type of transformation is applied to tables in the data warehouse that have an equivalent table in the operational schema. Multiple sources to one target transformation: This transformation translates the third normal form of the operational schema into the de-normalized form of the data warehouse by combining multiple source tables. I.e. in the operational schema transactions are usually normalized into lineitems and orders. In a dimensional data warehouse this relationship is de-normalized into a single fact table, basically materializing the join between lineitem and orders. One source table to multiple targets transformation: This transformation is the least common and occurs if, for efficiency reason, the schema of the operational system is less normalized than the data warehouse schema. Step 3 properly

manages data that is subject to historical retention (i.e., slowly changing dimensions). Finally, Step 4 inserts the new fact records and deletes fact records by date.

5.1 About Business Keys and Surrogate Keys

Contrary to OLTP systems, decision support systems typically use surrogate keys as primary keys to link fact tables to dimensions. They are usually generated by the database management system in form of sequential numbers (SEQUENCE) and not derived from any application data in the database. This has many advantages for schema management, performance and historic data management. While preserving uniqueness surrogate keys protect the database relationships from changes in data values or database design. Surrogate keys are generally composed of compact data types that might increase performance. Mostly, surrogate keys are used to preserve historic data in dimensions by making the current dimension entry a historical entry and adding a new dimension entry for the most current values.

Each dimension in the data warehouse uses a surrogate key as its primary key, while the tables in the source schema use traditional primary keys, also referred to as business keys. In addition to the surrogate key, each dimension contains the business keys of the corresponding entity of the operational source schema. This is necessary so that updated rows and transactions from the operational schema can be mapped to existing data in dimensions. Additionally, each dimension that retains historical information contains two date fields, `rec_start_date` and `rec_end_date`, to indicate the date range for which dimension entries are valid. The most current entry in a history keeping dimension uses a NULL value in the `rec_end_date` column. In order to facilitate the mapping of business keys to surrogate keys both in fact and dimension tables and the relationships between tables of the operational schema and the data warehouse schema, TPC-DS provides views defining the mapping of source schema tables to target schema tables.

```
CREATE VIEW item_view as
SELECT next value for item_seq i_item_sk
      ,item_item_id
      ,current_date i_rec_start_date
      ,cast(NULL as date) i_rec_end_date
      ,item_item_description
      ,item_list_price
      ,item_wholesale_cost
      ,i_brand_id ,i_brand
      ,i_class_id ,i_class
      ,i_category_id ,i_category
      ,i_manufact_id ,i_manufact
      ,item_size i_size
      ,item_formulation
      ,item_color
      ,item_units
      ,item_container
      ,item_manager_id
      ,i_product_name
FROM s_item
LEFT OUTER JOIN item ON (item_item_id = i_item_id
                     and i_rec_end_date is null);
```

Figure 12: Item View

As an example Figure 12 shows the Item view, which represents the data to be loaded into the Item dimension. The refresh data set is represented by the table `s_item`. In order to find the surrogate keys corresponding to the business keys in the `s_item` table, the two tables are joined on the `item_id` (business key). Since as in real systems, data in the data warehouse can be somewhat out of sync with the operation system, the tables are joined with an outer-join. In order to obtain the most recent surrogate key the additional predicate `i_rec_end_date is null` is necessary. Since Item is a Type 2 dimension (see next sections), a new surrogate key is generated by calling the SQL construct `next value` for the item sequence (`item_seq`). The `i_rec_start_date` is set to the current date while the `i_rec_end_date` is set to null indicating the most recent entry.

Apart from modeling the behavior of real life systems, the views test different code paths, such as the management of sequences, outer-joins and index maintenance. Especially for fact tables the execution of these views can be very resource intensive since the source data contains large amounts of data. Although DBMS specific, indexes on the business keys are probably the most efficient way of executing this join.

```
CREATE VIEW ssv AS
SELECT
 d_date_sk ss_sold_date_sk,
 t_time_sk ss_sold_time_sk,
 i_item_sk ss_item_sk,
 c_customer_sk ss_customer_sk,
 c_current_cdemo_sk ss_cdemo_sk,
 ...
 (i_current_price-plin_sale_price)*plin_quantity ,
 plin_sale_price * plin_quantity,
 i_wholesale_cost * plin_quantity,
 i_current_price * plin_quantity,
 i_current_price * s_tax_percentage,
 ...
FROM
 s_purchase left outer join customer on
                (purc_customer_id=c_customer_id)
            left outer join store on
                (purc_store_id=s_store_id)
            left outer join date_dim on
               (cast(purc_purchase_date as date)
            left outer join time_dim on
                (PURC_PURCHASE_TIME = t_time),
 s_purchase_lineitem
            left outer join promo on
                (plin_promotion_id = p_promo_id)
            left outer join item on
                (plin_item_id = i_item_id)
WHERE purc_purchase_id = plin_purchase_id
  AND i_rec_end_date is NULL
  AND s_rec_end_date is NULL;
```

Figure 13: Store Sales View

Inserting new data into fact tables comprises of two tasks (see Step 1 and Step 2 in Figure 17). First, data from the refresh data set is joined to dimension tables to swap the business keys from the operational system with the most current surrogate keys from the data warehouse. Additionally, since the operational system normalizes data for sales fact tables into two tables, Lineitem and Orders, these two tables need to be joined (see Step 1 in Figure 11). As an example Figure 13 shows the view for the Store_Sales fact table. Because of space constraints some of the columns are omitted. The fact table rows are built by joining two tables from the operational system: `s_purchase` and `s_purchase_lineitem` on their purchase id (see `where clause`). In order to obtain the surrogate keys for the dimensions time, date, item, customer and customer_demographics, these rows are subsequently joined using outer-joins on their business key to their corresponding dimensions. Finally, there are two additional predicates on the `rec_end_date` for the Type 2 dimensions.

5.2 Fact Table Maintenance

Fact tables hold business facts (a.k.a. measures) and foreign keys (surrogate keys), which refer to dimension tables. The business facts are collected in the operational system of a company as order occur, resulting in large amounts of data to be inserted and deleted periodically from fact tables. Corresponding to these two operations, TPC-DS defines insert and delete data maintenance functions for fact tables.

Step 2 (see Figure 17) inserts the data from the view into the fact table. Depending on the DBMS implementation this operation can be implemented as an insert with append resulting in bulk inserts at the end of the database.

```
insert into store_sales (select * from ssv);
```

Figure 14: Store Fact Table Insert Operation (Step 2)

The Algorithm of deleting old data from fact tables depends on the type of fact table. Data is deleted from sales and the inventory tables by specifying sales date ranges. For instance, "delete all sales that occurred between data x and date y. Returns can occur within three months of their sales, therefore, return data cannot be deleted by simply specifying date ranges in the returns table. Instead deletes of return data are triggered by deletes of their corresponding sales (see Figure 17).

```
DELETE FROM store_sales
WHERE ss_sold_date_sk IN
     (SELECT d_date_sk
      FROM s_del_date_m,date_dim
      WHERE d_date BETWEEN Begin and End));
```

Figure 15: Store Sales Delete Data Maintenance Function

```
DELETE FROM store_returns
  WHERE sr_ticket_number IN
   (SELECT ss_ticket_number FROM store_sales
    WHERE ss_sold_date_sk IN
     (SELECT d_date_sk FROM s_del_date_m,date_dim
      WHERE d_date BETWEEN Begin and End));
```

Figure 16: Store Return Delete Data Maintenance Function

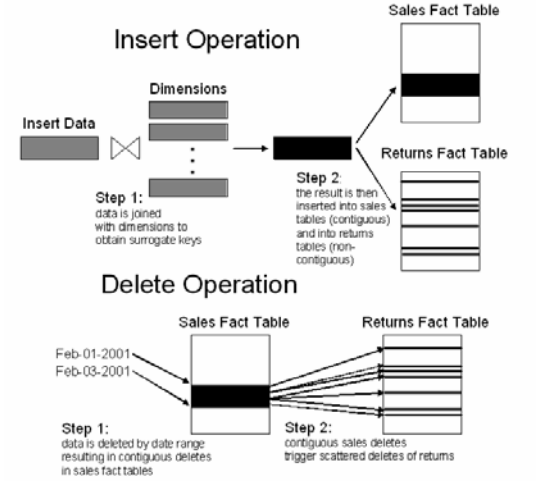

Figure 17: Logical View of Insert and Delete Operations

The Delete and Insert operations complement each other. That is, the same amount of data that is deleted is inserted into the same date range. This guarantees that the first and the second runs of the Performance test are executed against the same amount of data. The intention of these operations is to exercise both range and scattered deletes. For instance, the number of sales data blocks that need to be accessed may be minimized if sales data is clustered on sales date ranges. In the extreme case if data is finely partitioned by date, a partition drop operation can accomplish an entire delete operation. Contrary, the return delete operation are always scattered since returns can occur in a three month window after their corresponding sales.

5.3 Dimension Maintenance

Dimension tables contain attributes that further detail the business facts stored in fact tables. They are used to constrain and group data during complex queries. Since dimensions contain attributes that may change over time, Kimball [10] refers to them as Slowly Changing Dimensions (SCD). Among the four methodologies of dealing with the management of SCD TPC-DS concentrates on Type 0, 1 and 2.

Type 0 is used for those dimensions that never or very infrequently change data. In TPC-DS; that is, no Type 0 data changes during the benchmark after the initial load. Date, Time and Reason are examples for Type 0 dimensions.

Existing data in Type 1 dimensions (non-history keeping) is overwritten with new data, and therefore this type of dimension does not track historical data at all. This is most appropriate when correcting certain types of data errors, such as the spelling of a name. TPC-DS defines the algorithm for dealing with Type 1 dimensions as follows: For every row in the refresh data set the corresponding row in the data warehouse dimension needs to be identified. This is done by comparing the business key of the refresh data set with the business key of the data warehouse dimension. Then all non primary key columns are updated with the new data. An example of a Type 1 dimension in TPC-DS is Promotion.

Type 2 dimensions (history keeping) track historical data by creating multiple records with separate keys. The algorithm of maintaining Type 2 dimensions is slightly more complex then that of Type 1 dimensions because every entity may be represented in the data warehouse with multiple rows, i.e. historical entries. Hence, for every row in the refresh data set the business keys must match and the `rec_end_date` column must be NULL, corresponding to the most recent existing record. After the `rec_end_date` column of the matching row is set to the current system date a new row is inserted with the data in the refresh data set and a system generated new surrogate key. An example for a history keeping dimensions is the Item table.

```
UPDATE item set i_rec_end_date = SYSDATE
WHERE  i_item_id IN (SELECT i_item_id FROM itemv)
   AND i_rec_end_date IS NULL;

INSERT INTO item (SELECT * FROM itemv);
```

Figure 18: Item Data Maintenance Function

Figure 18 shows one implementation of the data maintenance function for the Type 2 dimension Item. It consists of two steps. In the first step the `i_rec_end_date` of those rows, which contain the current dimension version, are set to the current system date. This is done with an UPDATE statement combined with an IN clause selecting all rows from the item view (see Figure 12). This causes random reads from the dimension tables, as the rows to be updated are not contiguous. In the second step all rows from the item view are inserted into the item dimension. If no clustering is defined on the item dimension, this step can be performed as an append making this a bulk insert.

6. METRIC ANALYSIS

Because TPC benchmark publications are compared by their primary metric, it is important to the success of a TPC-DS that its primary metric is well understood, be aligned with typical DSS businesses and be "unbreakable". Undoubtedly, the primary metric defines the focal point of the performance work of any given benchmark publication. It is the intention of benchmark to define this focal point to encourage performance tuning in certain areas. Hence, only those areas of the workload that give the largest "return of investment" will be tuned and those that don't will be neglected. Ultimately, the goal is to increase the primary metric to beat the competition. By unbreakable we mean that the use of no single technology can be exploited to disproportionably increase the metric; this occurred with the primary metric of the former TPC-D benchmark when materialized views were introduced at the end of the nineties.

Since no elapsed times for TPC-DS are publicly available we analyze the impact of various performance tuning scenarios by calculating TPC-DS's primary metric using elapsed times from a published 100GByte TPC-H benchmark. We assume the following statistics: with minimal auxiliary data structures, as allowed by TPC-H (indexes on primary/foreign keys and date columns and horizontal partitioning) and run sequentially there are 32 percent short running queries (4s), 50 percent medium long running queries (17s) and 18 percent long running queries (102s); the load time is 3600 seconds; and one execution of all data maintenance functions is 390 seconds. Using these elapsed times, the different elements of TPC-DS metric can be calculated as:

T_{QR1}=TQR2=S*99*(0.32*4+0.5*17+0.18*102)=S*2785.86

T_{DM}=S*390, T_{Load}=3600

S stands for the number of streams, e.g. concurrent users

The scenarios are defined as: Scenario 1 analyses the impact of performance tuning on the 3 major parts of the benchmark, namely Database Load, Query Run and Data Maintenance (ELT); Scenario 2 shows how the use of materialized views can impact the metric; Scenario 3 shows the impact the number of streams (number of simulated users) has on the metric.

6.1 Scenario 1: Performance Tuning on Database Load, Queries and Data Maintenance

In this test we analyze the impact to the metric of performance improvement in the three disciplines of the TPC-DS workload: Database Load, Queries and Data Maintenance. Figure 19 shows one graph for the Database Load and one for Data Maintenance. The Query discipline is subdivided into three graphs, one each for short, medium and long running queries. Each graph shows the impact of performance improvements in each of the disciplines to the primary metric (QphDS) as a percentage of our initial assumption. 100% means no performance improvements and 90% means 10% performance improvements, etc. No side affects of performance improvement are considered. That is, only pure code-length or hardware improvements, e.g. CPU speedup are considered. No performance improvements such indexes or materialized views are considered that might have an impact to the elapsed times of other phases of the benchmark, i.e. Load or Data Maintenance. Hence, the metric can be expressed as follows:

$$QphDS@SF = SF * 3600 \frac{2}{2(0.32*4+0.5*17+0.18*102)+360+36}$$

The first graph in Figure 19 (diamond) shows that improving the load (T_{Load}) down to ten percent of its original elapsed time (from 3600s to 360s) has a marginal effect on the primary metric. In this scenario we use seven concurrent users (streams). Hence, only seven percent of the load time is accounted for in the primary metric. Similarly, improving short running queries or the data maintenance (small and large square) show little impact if improved down 10 percent of their original elapsed times. However, improving medium or long running queries (triangle and cross graphs) have a large impact to the metric. This is an expected behavior of the arithmetic mean. However, note that in this scenario performance improvements of queries are assumed to have no impact on the elapsed times of the database load and data maintenance. That is, they are assumed without the use of auxiliary data structures but with performance improvements such as optimizer enhancements or faster hardware. It is assumed that performance improvements of this magnitude are not likely to occur within the near future without the use of auxiliary data structures. Also, note that the larger the scale factor has a higher minimum number of streams, which will amplify the impact of the Database Load and Data Maintenance parts of the workload.

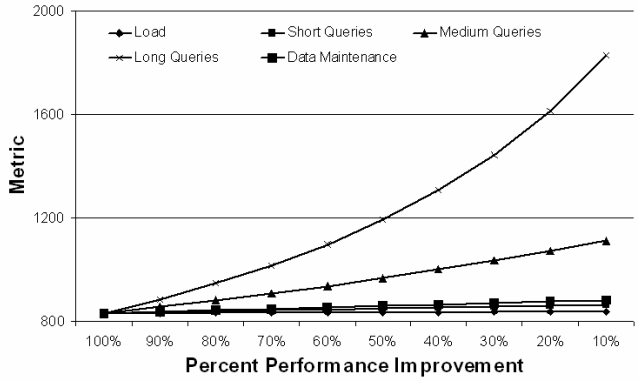

Figure 19: Improving Load, Queries and Data Maintenance

6.2 Scenario 2: Use of Materialization

Materialization techniques commonly used in modern decision support systems to speed up frequently used reporting queries. However, an unconstrained use of these techniques could potentially dramatically inflate the metric. As explained in sections 3 and 4, the use of materialization is restricted to the catalog sales channel. In this scenario, as an upper bound, we analyze the impact of materializing all reporting queries by reducing their elapsed time to one second. The cost for their materialization is counted in the load time and data maintenance phase as follows: The elapsed time for creating one materialized view per query is counted in the database load. The elapsed time is the original query elapsed time times a factor expressing their additional cost, referred to as the materialization cost factor (MCF), which we vary between 0.25 and 10 to capture a variety of cost models. An MCF of 0.25 signifies a materialization overhead of 25 percent and 10 signifies a 10 x overhead. The same overhead is added to the data maintenance phase.

Figure 20 shows the impact of materializing all reporting queries and amortizing their cost by running up to 100 streams. For each of the cost factors (MCF) the metric first increases as the number of streams increases. However, since the Load and Data Maintenance costs also increase with the number of streams, the metric decreases with very large number of streams. The maintenance costs of the materialized views or summary tables (MCF) will determine how steep the metric increases and at how many

streams it peaks. The higher the MCF value is the slower is the increase and the sooner is the peak.

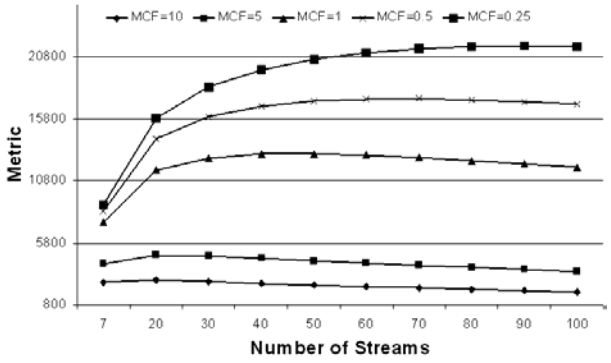

Figure 20: Metric Increase with Materialization

6.3 Scenario 3: Increasing Number of Streams

In this test we analyze the impact of running a TPC-DS benchmark with various numbers of streams. While the minimum number of streams for a given scale factor is set by the TPC-DS specification, the maximum number of streams is not limited. Assuming that one query stream, when run in isolation, takes T_1 seconds and that the systems scales linearly with the number of streams then, executing S streams takes S*T1 seconds. Additionally, assuming that the data maintenance functions scale linearly, i.e. each data maintenance, run on any update set, takes the same amount of time, e.g. T2, then S executions take S*T2. Hence, we can rewrite the metric canceling out the stream variable S from the numerator and denominator:

$$QphDS@SF = SF * 3600 \frac{198}{(2 * T_1 + T_{DM} + 0.01 * T_{Load})}$$

This metric is invariant from the number of streams. This is an important characteristic of the metric, because if the system scales linearly one should not be able to simply run with more streams to improve the performance metric. However, if the system scales super-linearly, i.e. it can schedule query execution such that queries in different streams can benefit of each other, e.g. piggybacking on IO patterns, join operations or intermediate results, then the metric can be increased. This type of optimization is both legal and desirable.

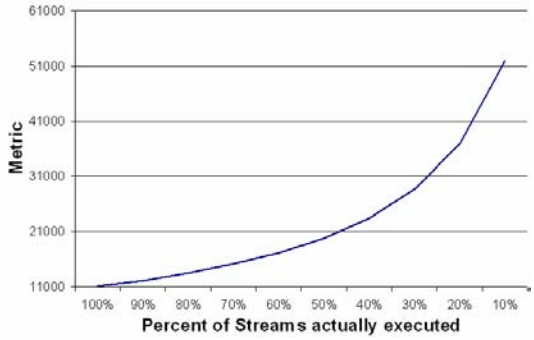

Figure 21: Metric Increase with Number of Streams

Figure 21 shows how the metric increases when the actual number of executed streams is reduced by a certain percent. That is, the first data point at 100% executes all streams, while at 50% the system only needs to execute 50% of the streams.

7. CONCLUSION

The TPC-DS benchmark is expected to be the next generation industry standard decision support benchmark, eventually replacing TPC-H. The benchmark is in the Formal Review phase.The TPC expects to receive comments and feedback from the industry and academia. The TPC-DS committee will review every comment carefully as this benchmark will be used by the hardware vendors and database vendors to demonstrate their capabilities and by customers as an important factor in purchase decisions.

As with prior TPC benchmarks, this benchmark workload will help accelerate development of hardware and database technologies to satisfy the requirement of modern data warehouse applications and also encourage research and development in optimization techniques in highly complex workloads.

8. ACKNOWLEDGMENTS

The authors would like to acknowledge Mike Nikolaiev, Ray Glasstone, David Adams, Bryon Georgson, Murali Krishna, Umesh Dayal and Christopher Buss for their comments and feedback and the members of the TPC-DS committee, especially Vincent Carbon, Susanne Englert, Douglas Inkster, Mary Meredith, Sreenivas Gukal, Doug Johnson, Lubor Kollar, Murali Krishna, Robert Lane, Larry Lutz, Priti Mishra, Juergen Mueller, Robert Murphy, Doug Nelson, Ernie Ostic, Gene Purdy, Haider Rizvi, Bryan Smith, Eric Speed, Cadambi Sriram, Jack Stephens, John Susag, Tricia Thomas, Kwai Wong and Guogen Zhang.

9. REFERENCES

[1] John M. Stephens, Meikel Poess: *MUDD: a multi-dimensional data generator*. WOSP 2004: 104-109

[2] Meikel Poess, John M. Stephens: *Generating Thousand Benchmark Queries in Seconds*. VLDB 2004: 1045-1053

[3] Meikel Poess, Raghunath Othayoth: *Large Scale Data Warehouses on Grid: Oracle Database 10g and HP ProLiant Systems*. VLDB 2005: 1055-1066

[4] Meikel Poess, Bryan Smith, Lubor Kollár, Per-Åke Larson: *TPC-DS, taking decision support benchmarking to the next level*. SIGMOD Conference 2002: 582-587

[5] Meikel Poess, Chris Floyd: *New TPC Benchmarks for Decision Support and Web Commerce*. SIGMOD Record 29(4): 64-71 (2000)

[6] Michael Stonebraker et. al.: C-Store: A Column-oriented DBMS. VLDB 2005: 553-564

[7] Naveen Reddy, Jayant R. Haritsa: Analyzing Plan Diagrams of Database Query Optimizers. VLDB 2005: 1228-1240

[8] Public release of TPC-DS (v0.32) preliminary draft: http://www.tpc.org/tpcds/default.asp.

[9] Raghunath Othayoth, Meikel Poess: *The Making of TPC-DS.* VLDB 2006: 2046-1058

[10] Ralph W. Kimball, Warren Thornthwaite, Laura Reeves and Margy Ross: *The Data Warehouse Lifecycle Toolkit*, New York, NY: John Wiley and Sons, 1998

[11] Pricing http://www.tpc.org/pricing/spec/ Price_V1.0.1.pdf

[12] TPC-D Version 2.1: http://www.tpc.org/tpcd/default.asp

[13] TPC-DS Draft Version: http://www.tpc.org/tpcds/tpcds.asp

[14] TPC-H Version 2.6.0: http://www.tpc.org/tpch/default.asp

[15] US Census Bureau, *Unadjusted and Adjusted Estimates of Monthly Retail and Food Services Sales by Kinds of Business:2001*, Department stores (excl.L.D) 4521.

[16] William H. Inmon: *EIS and the Data Warehouse, Data Base Programming/Design*, November 1992.