# Generating Thousand Benchmark Queries in Seconds

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

John M. Stephens, Jr.
Gradient Systems
643 Bair Island Road #103
Redwood City, CA-94063
650-566-9380
jms@gradientsystems.com

## ABSTRACT

The combination of an exponential growth in the amount of data managed by a typical business intelligence system and the increased competitiveness of a global economy has propelled decision support systems (DSS) from the role of exploratory tools employed by a few visionary companies to become a core requirement for a competitive enterprise. That same maturation has often resulted in a selection process that requires an ever more critical system evaluation and selection to be completed in an increasingly short period of time. While there have been some advances in the generation of data sets for system evaluation (see [3]), the quantification of query performance has often relied on models and methodologies that were developed for systems that were more simplistic, less dynamic, and less central to a successful business. In this paper we present QGEN, a flexible, high-level query generator optimized for decision support system evaluation. QGEN is able to generate arbitrary query sets, which conform to a selected statistical profile without requiring that the queries be statically defined or disclosed prior to testing. Its novel design links query syntax with abstracted data distributions, enabling users to parameterize their query workload to match an emerging access pattern or data set modification. This results in query sets that retain comparability for system comparisons while reflecting the inherent dynamism of operational systems, and which provide a broad range of syntactic and semantic coverage, while remaining focused on appropriate commonalities within a particular evaluation process or business segment.

## 1. INTRODUCTION

The number of different queries executed on production systems far exceeds the practical number of queries that can be covered in a benchmark specification. Hence, a major task in designing a data warehouse benchmark is to create a query set that both represents the real world and executes in a reasonable amount of time. TPC-D [7], as the first industry-standard benchmark, defined 17 complex, industry relevant SQL queries. While it relied on some broad simplifying assumptions, it represented a major step forward from the simple, home-grown query workloads that had been used for early characterization efforts. As query functionality improved, its successors, TPC-H and TPC-R [4], added 5 queries to keep pace. TPC's next generation DSS benchmark, TPC-DS [5], is anticipated to employ more than one hundred queries. Due to increased database system functionality (e.g., SQL99 with OLAP extensions), the number of representative SQL query combinations on any particular schema is still very large. With an ever-increasing number of queries comes a need for a fast, reliable, extensible query generator.

DBMS functionality has improved dramatically since the TPC introduced the first industry-standard DSS benchmark in 1995. Today's systems rely heavily on sophisticated cost based query optimizers to generate the most efficient query execution plan. A DSS benchmark that evaluates optimizers must both provide the rich data set details on which they rely (uniform and non-uniform distributions, data sparcity, etc.), and must also test the optimizer's capability to generate the most optimal plan under all circumstances. Functionality to partition a data set and limit IO activity to that portion of the global data set that is of interest to a particular query is now commonplace. Traditional SQL operators have been buttressed by new semantics, which allow business processes to be closely modeled in query syntax (i.e., MDD and OLAP), and novel access patterns that allow the use of data sampling and pre-computation to speed answers to common business questions (i.e., summary tables and join indexes). With increasing sophistication and complexity available in even commodity systems, a successful benchmark must provide a transparent and adaptive architecture that allows meaningful comparisons across different systems, while allowing easy adaptation to emerging technologies.

As DSS have become an increasingly common and important piece of successful IT infrastructure, the focus and sophistication that vendors apply to benchmarks and their tuning has increased significantly. To compare the performance of different systems, it is essential that all systems run the same workload under the exact same execution rules. At the same time, it is important that the person conducting the test be allowed to tune the system appropriately. The challenge is to assure that the benefits that arise from pre-benchmark tuning are appropriate to the environment being evaluated and representative of

**Proceedings of the 30th VLDB Conference,**
**Toronto, Canada, 2004**

improvements that can be achieved outside of a benchmark situation (i.e. in a production system).

Finally, the operational environment for query systems has also changed. There is still a significant set of queries that is run in a batch environment, and can appropriately be subjected to highly specialized tuning. Other queries, however, are not known in advance except in broad terms, giving less possibility to optimally tune the system (ad-hoc queries To properly characterize both query approaches, a benchmark needs to acknowledge the different operational assumptions of each methodology. In cases where a batch query environment is being modeled, it is acceptable to provide well known, largely static queries, since it is assumed that the target operational environment would subject the queries to the same level of high-specialized tuning that would inevitably result in a competitive benchmark setting. To model an ad-hoc environment, the amount of fore knowledge of a particular query's phrasing needs to be limited. Regardless of the predominance of static or ad-hoc queries, a benchmark methodology must provide consistent, verifiable, comparable results.

Each of these technologies and enhancements can be exploited to provide an unrealistic vision of system performance, unless benchmarks are careful structured to realistically reflect the enhancement provided, without allowing unreasonable or inappropriate over optimization. QGEN addresses many of these concerns. For instance, if all query substitutions are known prior to benchmark execution, the optimizer can be tuned only for those predicates while ignoring the more general cases. If data access patterns are known prior to benchmark execution, the system under test (SUT) can be tuned for in a non-realistic way and system weaknesses can be easily covered up by, for instance, only creating auxiliary structures for part of the data set. QGEN's linkage to the underlying data generation assures appropriate coverage of the whole data set. Most DSS benchmarks include a multi user test (e.g. TPC-H/ TPC-R) in which multiple concurrent sessions execute queries simultaneously. Executing the same queries across multiple sessions is not desirable because one could easily implement a feature that materializes the result of every new query. Subsequent sessions executing these queries can take advantage by simply displaying the content of the previously materialized data without computing any results themselves. QGEN's ability to dynamically parameterize a query set minimizes these risks. Its ability to create query sets which are extensible and random, but statistically comparable, , makes it possible to construct a query set that retains query characteristics without resorting to blind query repetition. This removes the need for repetition that is not representative of the query environment being modeled.

The remaining sections of this paper review the evolution of query models culminating in a detailed description of the SQL query generator, QGEN, which was developed by the TPC for generating queries in TPC-DS. Before discussing the general concepts behind QGEN we will briefly introduce the data and query models, which motivated its development. Prior to concluding we will show empirically that QGEN generates comparable queries using examples from TPC-DS's query set.

## 2. RELATED WORK

In [1] Slutz presented RAGS, a system to stochastically generate valid SQL statements. This system has been employed at Microsoft for deterministic testing of SQL statements. For a specific database schema RAGS generates syntactically correct SQL statements by walking a stochastic parse tree. As RAGS can quickly generate and execute millions of statements to increase the coverage of system functionality testing, it is not quite suitable to generate queries to test system's performance. For benchmark purposes it is not desirable to completely randomly generate queries since their execution times cannot be predicted or limited, especially, because of the large query set that needs to be executed to achieve good coverage of performance characteristic.

## 3. EXISTING QUERY MODELS

### 3.1 STATIC QUERY MODELS

The simplest approach to benchmarking is to record pre-existing events or behaviors and simply replay them under controlled conditions. Early vendor benchmarks employed a similar approach. Sample queries were captured from a production system (or crafted to match an intended implementation) and executed against potential solutions to provided comparative performance data.

This approach has some obvious and attractive benefits. The testing methodology is easily understood and, since the queries are completely static, comparability between benchmark executions is guaranteed. Unfortunately, the shortcomings of this approach are equally compelling. The functionality coverage provided by the benchmark is limited by the captured queries, the benchmark is unlikely to adapt well for evaluating changes to the schema or workload without significant intervention, and the results produced by a multi-user execution are likely to be misleading, as the limited and static nature of the query makes it easier to pre-compute its result, or to reuse the result that was computed in one user session to speed execution in another (i.e. multi user run).

```
SELECT sum(l_extendedprice)
FROM   lineitem
WHERE  l_shipdate >= '10-01-03'
AND    l_shipdate < '12-31-03';
```

**Figure 1: Static DSS Query**

The simple DSS type query in Figure 1 can be used to illustrate additional drawbacks of a static query model. It aggregates the extended price of all lineitems in the fourth quarter of 2003. A simplified materialized view, only covering the fourth quarter of 2003, turns this IO intensive query into a single row lookup query reducing the elapsed time to sub-second. If there is no other query that benefits from aggregation of lineitem data, there is no need to cover any more data than the fourth quarter of 2003. Similarly, other auxiliary structures such as indexes could only be created on a subset of data, for instance, using local indexes build on a subset of data, for instance by using table partitions. Assuming that the remaining data of 2003 is not accessed in any other query, another possible optimization is to horizontally partition the table into accessed and non-accessed sections. The non-accessed sections can then be moved to cheaper permanent storage, such as tape drives increasing performance and reducing the total cost of the system[1].

## 3.2 SIMPLE SUBSTITUTIONS

A completely dynamic query model model, in which the query text was created only at the time of query execution and little or none of the text was known to the benchmarker in advance would remove much of the potential for over-optimization, but would be likely to violate the most fundamental requirement for any benchmark: to impose a reproducible stimulus on the system under test. For a decision support benchmark with its focus on query execution, this means that if a query is executed multiple times under the same circumstances, the execution times from run to run do not vary significantly. TPC's original data warehouse benchmarks (TPC-D, TPC-H and TPC-R) fulfilled this requirement by assuming a very simple data model, (e.g., uniform distributions for all columns), and by constraining the scope of possible query-to-query variation. Figure 2 (see below) shows the template of TPC-H's Query 6. It uses three substitution variables, DATE (the first of January of a randomly selected year within [1993 .. 1997]), DISCOUNT (randomly selected within [0.02 .. 0.09]) and QUANTITY (randomly selected within [24 .. 25]).

```
SELECT  SUM (l_extendedprice*l_discount)
FROM    lineitem
WHERE   l_shipdate>=date'[DATE]'
AND     l_shipdate<date'[DATE]'+interval'1'year
AND     l_discount between [DISCOUNT] - 0.01
                   and [DISCOUNT] + 0.01
AND     l_quantity < [QUANTITY];
```

**Figure 2: TPC-H's Query Template 6**

This approach is clearly an improvement over the purely static query model that it replaced. The risk of pre-computation is greatly reduced (controlled largely by the range of possible substitution values), the query set scales to multi-user execution without the risk of result reuse, and the lack of complete predictability provides a meaningful test of query optimizers. Further, by providing a standardized method for the execution and reporting of test results, the TPC benchmarks provided a more robust and trusted basis for system comparison. That the benchmarks were widely published is a testament to their success. They too had problems, however. The inflexible linkage between the query set and the data generator made it difficult to expand or alter the query set as DBMS technology improved, and the limited query breadth imposed by this increased maintenance burden left the benchmark vulnerable to over-optimization through specialized indexes and other data structures. While appropriate constraint of the configuration and execution rules of TPC-H have allowed it to remain viable, it became clear that the benchmark, with it simplified, third-normal data model and proscriptive implementation rules was not sufficient to capture the breadth of modern decision support systems performance.

## 4. NEW APPROACH TO QUERY MODELING

In an attempt to address some of the shortcomings of the simple substitution models used in the past, and to update its benchmark suite to align with common DSS methodologies, the TPC is developing a new benchmark, TPC-DS. The new benchmark is built around an updated data model, a more sophisticated concept of a decision support user, and a next generation query model, implemented in QGEN, the query generator for the new benchmark.

The examples in this paper use TPC-DS' data model. In line with modern data warehouse systems, QGEN and MUDD employ a variant of a star schema. The schema utilizes the business model of a large retail company having multiple stores located nation-wide. Beyond the brick and mortar stores, the fictitious company sells goods through catalogs and the Internet. Consequently, its operational processes include a store order and return system, a catalog order and return system, and a web order and return system. In addition, it provides an inventory system for all warehouses and a promotion system. Each system is represented in the schema as a snowflake with dimensions shared among all snowflakes. Dimensions such as Date, Store, Item and Promotion are arranged in the classical star constellation around Store Sales. Customer, Customer Address, Customer Demographics, Household Demographics and Income Band are arranged in a snowflake fashion. Apart from their relationships to Store Sales, Customer Address, Customer Demographics and Household Demographics have additional relationships to Customer. This allows the data model to capture both the classification of the customer at the time of the sale (through the links to the Sales fact tables) and at the time of

[1] Performance Analysis is often paired with a cost analysis. In every TPC benchmark publication includes a price analysis and in most benchmarks price performance gets reported as a metric.

the query (through the links to other dimensions). The Store Returns system shares all dimensions of Store Sales. Additionally, it introduces a new dimension describing the reason for the return (Reason).

The structure of the Catalog and Internet sales channels are identical to the Store Channel, except that some dimensions are renamed (e.g., store becomes catalog). Rows in the fact table store the numerical measurements of the business modeled. In our case, these are sales, returns (store, catalog and internet), inventory movements and promotions. Each of these subject areas is modeled with one fact table. It contains foreign keys to dimension tables and numerical measures (additive and non-additive).

Dimensions contain numerical surrogate primary keys and descriptive attributes to further describe the dimension. There are static and non-static dimensions. Non-static dimensions are maintained as part of an ETL process. Describing the details of this process exceeds the scope of this paper. Details can be found in [5].

The aforementioned star schema defines three hierarchies to allow for easy browsing of dimension data. Despite the normalization that led to the snowflake schema, each hierarchy is confined to one dimension. Each hierarchy is strict meaning that an entity in the lower level maps to exactly one entity of the higher level. For instance, one city maps to exactly one county, which maps to exactly one state.

The development of QGEN has been driven by the requirements of the TPC-DS query model. The benchmark characterizes the queries submitted to the SUT into one of four query classes, representing different kinds of database activity: reporting, ad-hoc queries, iterative enquiries and data extraction (e.g., data mining activities). In this query model, different degrees of complexity, variability and predictability of the queries submitted to the SUT are captured in the different query classes. Accordingly, the query generator needs to produce an arbitrarily large, random set of queries within the constraints of each class while meeting the reproducibility requirements for the benchmark as a whole.

## 4.1 QGEN

QGEN is a command-line utility that translates an arbitrary set of query templates into streams of fully-qualified, valid SQL. Based on a LALR(1) grammar that defines the template syntax, QGEN can quickly produce an arbitrarily large query set for any of a number of query classes.

Template-based queries are defined to be sets of one or more pseudo-random, valid SQL statements generated shortly before benchmark execution. Template-based queries are intended to model common, well-understood queries, which are generated in conjunction with periodic reports or common business tasks. It is assumed that the precise values or targets of a given instance of a template-based query is random, but that the general format and syntax for the query is tightly tied to a business process and the syntax is therefore largely predictable and well-known.

To make queries less "known" is to vary SQL predicates. QGEN defines a query template language, which allows for the different types of SQL predicates, found in today's decision support systems:

- single value.
- range,
- in-list, and
- like.

Queries resulting from the same template execute in an equivalent execution time, even though the underlying data set is highly skewed and non-uniform.

### *4.1.1 MUDD AND QGEN*

QGEN depends heavily on the underlying data set. A query generator can only uncover data relationships that exist in its target data population, and a query tool intended for cross-vendor comparisons must rely on the existence of a data set from which it is possible to produce comparable queries. It is the creation and manipulation of the data domains and distributions within the data set that makes the generation of meaningful and comparable query sets.

The data generator that is coupled with QGEN is called MUDD [6]. MUDD and QGEN represent an evolution in query benchmarking, because they have externalized the dependency between a query tool and its data generator into a set of external text files that define the data domains and distributions for both the data generation and query creation. This allows the modification of the data distributions and their use in the queries without requiring the recompilation of the underlying tool set.

The core of the domain and distribution functionality within MUDD and QGEN is the ability to define an arbitrary distribution of values. In addition to defining the values themselves, the tools rely upon the relative frequencies of the values (which may vary when the same value set is used in different ways) along with any related or correlated values. All of this information is contained in an ASCII file, so values and weightings can be altered during experimentation, and fine tuning of the benchmark can occur without requiring changes to the toolset itself.

### *4.1.2 GENERAL DEFINITION SYNTAX*

A query template is divided into two parts. The Substitution Declaration declares the substitution rules. They consist of a list of substitution tags. Each tag is declared as a substitution type (distribution) with specific substitution parameters. The so defined tags can be used in the SQL Text part of the query template, which consists of a SQL query, to specify the substitution values of query predicates. Each occurrence of the substitution tag in the SQL text is substituted according to the substitution type. Multiple occurrences of the same tag are substituted with the same value. However, if a substitution tag is post-fixed with a number, then each occurrences is substituted with a new value. The general syntax for a substitution rule is:

```
DEFINE <tag>=
              <substitution type>
             (<substitution parameters>);
<tag> = string[30];
<substitution type> = <RANDOM | DIST | TEXT>
```

Consider the following example T1 in Figure 3.

```
#SUBSTITUTION DECLARATION
DEFINE
      month = RANDOM (1, 5, UNIFORM);
DEFINE
      high_color=dist("colors",1,high);

#SQL TEXT
SELECT sum(S.salesprice)
FROM   store S, item I, time T
WHERE  T.sold_date between month
                        AND month+2
AND    I.color = high_color
```

**Figure 3: Example Query Template T1**

In its substitution declaration it defines two substitution tags, *month* and *high color*. They will be explained more in detail in the following sections. The SQL joins the fact table *stores* with the two dimension tables *item* and *time.* Its sums the sales of all items in a specific time window (first predicate) which are of a specific color (second predicate).

### *4.1.3 RANDOM SUBSTITUTION*

The *Random* substitution type allows templates to use randomly-generated integers in a inclusive range [min,max] using normal or uniform distributions. The specific syntax for the RANDOM substitution rule is:

```
RANDOM(<min>,<max>,<distribution>)
<distribution> = [NORMAL | UNIFORM]
```

The RANDOM substitution is very commonly used in data warehouse queries since it can be used to implement the very common time predicates. Usually decision support queries are time based using month, quarter or year as their window of operation. The designer of a query template must assure that the values picked for *min* and *max* fall within the targeted comparability zone (range in the data domain with a uniform distribution). Figure 4 shows a very simple example of a random substitution. It defines the tag *month*, which uses a uniform distribution between 1 and 5. This tag is then on the predicate on sold_date.

### *4.1.4 DIST SUBSTITUTION*

The *DIST* substitution type allows a template to use one of the arbitrary distributions defined in conjunction with its data generator (MUDD) through their shared distribution files (.dst, .idx), which provide step functions of arbitrary complexity and resolution. The specific syntax for a DIST substitution rule is:

```
DIST (<name>, <value set>, <weight set>)
<name> = name of the distributions as defined in .idx
<value set> =  1-based index for the value to be returned
               from the distribution tuple
<weight set> = 1-based index for the selected weighting
               from the distribution tuple
```

```
DEFINE color = TEXT (
          ("red", 10),
          ("blue", 80),
          ("green", 10);
DEFINE add_predicate = TEXT (
          ("and palette like "[color]%", 30),
          ("", 70)
          );
Select count(*) from products where color != [color]
          [add_predicate];
```

**Figure 5: TEXT Substitution Syntax**

The DIST substitution does not require the template designer to know the specifics of the data distribution, only the names of the distribution and its weight and value sets (e.g., “colors” or “leap_year_sales”).

### *4.1.5 TEXT SUBSTITUTION*

The TEXT substitution replaces a particular tag with one of a weighted set of ASCII strings. In its basic form, this is can be employed in a like clause to produce a wild card predicate such as: `column_name like "<string>%"`, providing a crude form of text searching as demonstrated in the first part of Figure 5.

The real power of the TEXT substitution lies in its ability to include both static text and additional substitution tags. Whenever QGEN employs a TEXT substitution, it traverses the selected text looking for additional substitution tags. These are evaluated in turn, and final static text replaces the TEXT substitution tag in the query template. An example of this behavior can be seen in the second example from Figure 5, where the initial substitution for [add_predicate] can result in an additional occurrence of the *palette* predicate.

#### *4.1.6 DATA POPULATION WITH COMPARABILITY ZONES*

Synthetic data generators face an inherent challenge. If the data is too synthetic (e.g., completely uniform distributions), it runs the risk of being rejected for not capturing the “interesting” attributes of a real data set. Conversely, if it employs data that is gathered from transactions or installations (“the real world”) it risks being of little or no value to researchers and benchmarks, since it can neither produce comparable workload results nor be scaled to answer interesting hypothetical questions. MUDD attempts to find an appropriate mix of these two endpoints. For a majority of its data, it employs traditional synthetic distributions, yielding uniformly distributed integers, or word selections with a Gaussian distribution. For a number of crucial distributions, however, the data generator relies on data from the real world to produce more realistic data sets.

Given the importance of skew in a data set, and the dominance of this particular skew in this particular data set, omission would clearly be a poor choice. Instead, the data set needs to be “adjusted”, introducing zones of comparability – essentially flat spots in the distribution – that can be used to provide both the variability and the comparability that the eventual user of the generated data requires. With MUDD’s flexible approach to comparability zones each column domain can define its own comparability zones. They can differ in size within the column domain and in number between column domains. However, in order to allow for a large number of possible substitutions, one needs to adjust the number of comparability zones so that the requirements for parameter substitutions of the application to be tested are met. There should be at least one comparability zone large enough to fit the largest range substitution. Depending on how many different range substitutions the applications calls for this size must be adjusted. The upper bound of any comparability zone is limited by the number of comparability zones that can fit into the column domain.

As an example for a distribution with comparability zones, consider the likelihood of retail sales throughout the calendar year. Census data shows that a dramatic proportion of sales occur during the year-end holiday season, often as much as 30% of annual sales within the last two months of the year (and much of that in the last half of December) [3]. Figure 6 shows the Census data for retail sales by month in th USA.

For date predicates TPC-DS requires range substitutions of at most 90 days (one quarter). Hence to mimic the sales distribution above, in each year we define three comparability zones; 1) January to July; 2) August to October; 3) November to December. The database we are using for the remaining of this paper contains 5 years of data each year following the same distribution. Domain values in the first zone occur with a low likelihood in the data set (low zone), domain values in the second zone occur with a medium likelihood and domain values in the third zone occur with a high likelihood in the dataset. The actual likelihood of occurrences in each zone may vary from domain to domain. However, MUDD guarantees that all domain values in one domain have the same likelihood.

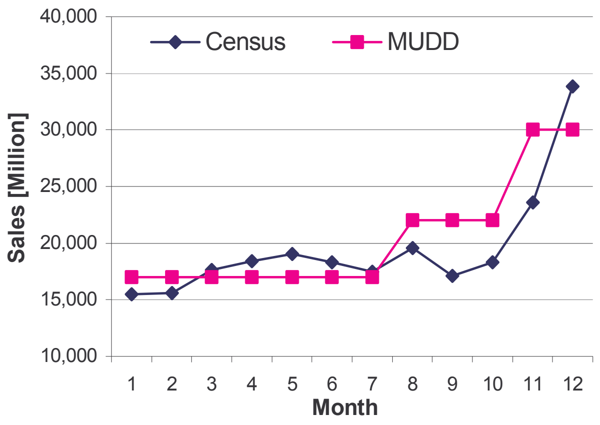


**Figure 6: Retail Sales by Month**

The fact that there is a correlation between month and sales amount is very imperative to this kind of column domain. If after ordering the domain values of a distribution the graph shows a step functions, that is, neighboring domain values (except for the edges) have the same occurrence likelihood, we define this as a Type A distribution. This characteristic enables the use of range predicates.

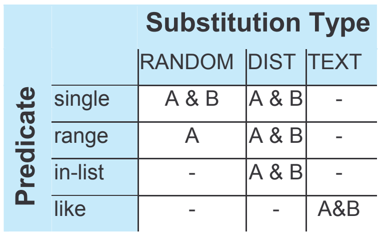
Clearly a skew this dramatic constitutes an important feature in the data set, and needs to be captured by the data generator. The use of the data needs to be considered too, however, if the year-end bulge were to be captured precisely, then it is likely that no two days would generate precisely the same sales volume. In our query example, it would then be impossible to assure that queries based on a date in this crucial range (or elsewhere throughout the year, for that matter) generated comparable amounts of load or activity on the system under test. The option left to the dataset/benchmark designer is to either remove queries based on dates in this region from the benchmark, or find some way to assure that they are comparable to one another.

The query designer needs to be aware of the comparability zones and write his queries accordingly. Consider the previous example. The first comparability zone contains 7x30=210 days. Assuming one would like to define predicates covering 90 days in this zone, the valid range for the left end point of all range predicates is 10-90=120 range substitutions of 90 days, the second zone allows for 1 substitutions, while the third zone is too small for any 30 day substitution.

MUDD defines other distributions that are not derived from the real world, and, most importantly, that do not constrain the relationship between domain values and their frequency. We define these as Type B distributions. Because of neighboring domain values not occurring with the same likelihood, range predicates are not allowed on Type B distributions. An example defines the domain of common colors (see Figure 9 and 10). The colors are sorted on their number of occurrences in the dataset. Similarly to the sales distribution, we define three comparability zones: low, medium and high. Colors are arbitrarily chosen to belong to any of the three comparability zones. Colors belonging to the low comparability zone occur less likely in the data set than colors of the medium zone. Colors belonging to the high comparability zone have the highest likelihood to occur in the dataset.

### *4.1.7 COMPATIBILITY OF SUBSTITUTION TYPES AND QUERY PREDICATES*

Due to their characteristics not all types of query predicates are supported by each substitution type with each type of distribution. Table 6 shows which substitution type is compatible with the different types of query predicates. Type A and Type B distributions can be used with the DIST and RANDOM substitution types for single value predicates. Range predicates can be used with Type A distributions and RANDOM. Using DIST range predicates can be used with both Type A and Type B distributions.



| | | Substitution Type | | |
|---|---|---|---|---|
| | | RANDOM | DIST | TEXT |
| Predicate | single | A & B | A & B | - |
| | range | A | A & B | - |
| | in-list | - | A & B | - |
| | like | - | - | A&B |

**Figure 7: Substitution Types Compatibility Matrix**

### *4.1.8 IMPLEMENTATION OF COMPARABILITY ZONES IN QGEN*

The core of the domain and distribution functionality within MUDD and QGEN is the ability to define an arbitrary distribution of values. In addition to defining the values themselves, the tools rely upon the relative frequencies of the values (which may vary when the same value set is used in different ways) along with any related or correlated values. All of this information is contained in Distribution Configuration (ASCII file), so values and weightings can be altered during experimentation, and fine tuning of the benchmark can occur without requiring changes to the toolset itself.

The format of a distribution definition is summarized in Figure 8 and 9. The result is a step function of arbitrary complexity and resolution. Two examples of common usage are provided. In Figure 7, two weights are provided – one to cover both leap and non-leap year. In Figure 9, common color names are grouped into classes. By selecting the appropriate weight set, it is possible to produce a weighted distribution across the entire color spectrum, or to randomly select a color from within a given class. The distribution example also demonstrates the use of multiple entries within the value set to provide correlated attributes to be selected from the distribution. In this case, the constituent primary colors are available for each listed entry.

```
SALES =
(1, "Jan 1"; 10, 10)
(2, "Jan 2"; 10, 10)
[…]
(59, "Feb 28"; 10, 10)
(60, "Feb 29"; 0, 10)
(61, "Mar 1"; 10, 10)
 […]
(348, "Dec 13"; 30, 30)
(349, "Jan 14"; 50, 50)
(350, "Jan 15"; 50, 50)
```

**Figure 8: Sales Distribution**

```
COLORS =
("Red", "Red", "None"; 100, 1, 0, 0)
("Blue", "Blue", "None"; 100, 1, 0, 0)
("Yellow", "Yellow", "None"; 100, 1, 0,
0)
("Green", "Blue", "Yellow"; 50, 0, 1, 0)
("Purple", "Red", "Blue"; 50, 0, 1, 0)
("Orange", "Red", "Yellow"; 50, 0, 1, 0)
("Taupe", "Orange", "Green"; 10, 0,0, 1)
("Mauve", "Purple", "Blue"; 10, 0, 0, 1)
("Pink","Yellow","Green"; 10, 0, 0, 1);
```

**Figure 9: Color Distribution**

### *4.1.9 PRESERVATION OF COMPARABILITY ZONES*

So far we have seen how step functions can be defined on single table column distributions to assure comparable queries in the presence of substitutions. But what happens to comparability zones if predicates are defined on multiple columns or if tables are joined? MUDD prohibits correlation between intra table column distributions and between join columns. That is all comparability zone distributions must be statistically independent. For instance, it is disallowed to define item color and size distributions such that green items are larger than blue items or sales distributions such that red items are more likely to be sold in December than in March. This is very important since comparability zones need to be preserved in the presence of multiple predicates and joins. If indeed there is no correlation between columns, choosing predicates in comparability zones of multiple columns still yields comparable queries. This is true for joins as well if there is no correlation of the join columns.

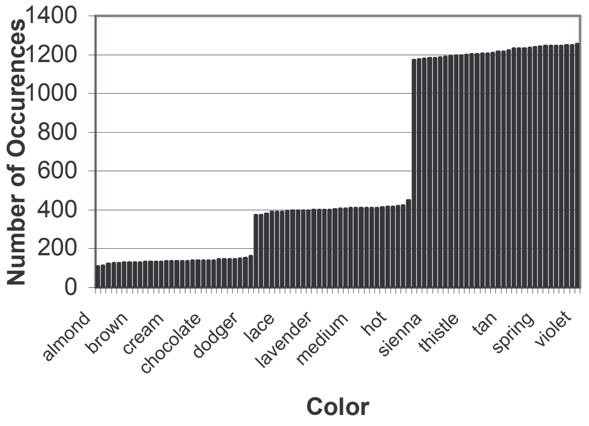


**Figure 10: Color Distribution**

## 4.2 EXAMPLE AND EXPERIMENTAL RESULTS

In this section we will present some experimental results of queries executed against a database demonstrating that QGEN indeed generates comparable query sets. We will discuss the example shown in Figure 3 in more detail. First, lets create a variant T1a of T1, which contains only one substitution parameter, high_color (see T1a)

```
#SUBSTITUTION DECLARATION
DEFINE
      high_color=dist("colors",1,high);
#SQL TEXT
SELECT sum(S.salesprice)
FROM   store S, item I, time T
WHERE  I.color = high_color
```

**Figure 11: QGEN Query Template T1a**

The existing benchmarks TPC-H and TPC-R have already shown that substitution parameters on uniform data distributions yield comparable queries. Since the parameter substitution model of QGEN operates on comparability zones, which have uniformly distributed data, it yields to comparable queries. To demonstrate this, we generate 10000 queries from templates T1a and T1 using QGEN. All queries of template T1a use substitution parameters for colors that are of the high likelihood comparability zone. Then we run these queries sequentially against a TPC-DS database and collect the elapsed times. In addition to the color substitution parameters of T1a, T1 uses a date predicate on sold_date (sold_date between *month* AND *month*+2) of the low season in the sales distribution (January to July). We also generate 10,000 queries using QGEN, execute them sequentially and collect the elapsed times. The y-axis in 11 shows the deviation from the mean of the elapsed time of these queries. The deviations are graphed in sorted order. The label on the x-axis shows the query run number.

Elapsed times for queries generated from the T1a template deviate between –0.5% and 0.5% from the mean. The Coefficient of Deviation for this dataset is 0.00177, well within the requirements. The second graph shows the elapsed time deviation to the mean of queries generated from the T1 template. They differ from about -0.5% to about 1%. This translate into a Coefficient of Deviation is 0.00317, which is also well within the requirements.

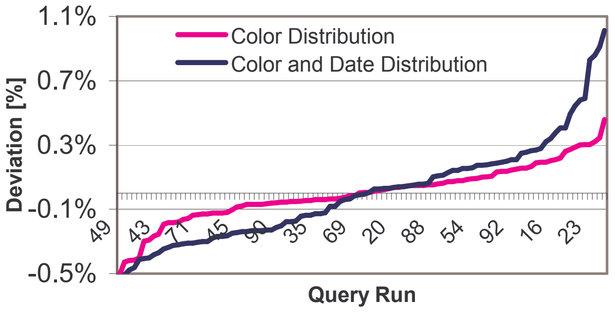


**Figure 12: Deviation from Men for Queries generated with T1 ad T1a Query Templates**

These two examples show that when multiple substitution parameters, which in isolation yield comparable queries, are combined in one query template the resulting query template yields comparable queries if the distributions of which the predicates are defined, are independent. For our example this means that there must not be a correlation between sales date and item color.

## 5. FUTURE DIRECTIONS FOR QUERY MODELS

One of the goals of TPC-DS is to generate comparable queries. This is guaranteed by comparability zones and by careful selection of parameter substitutions for selected predicates.

As query modeling embraces an increasingly interactive approach (i.e., OLAP), the requirement for individual query comparability will give way to comparability on final metric level. As business intelligence moves beyond basic SQL (i.e., into cubes or other complex data structures), even complex queries can provide sub-second response time. As a result it becomes possible to execute thousands of queries while keeping the benchmark execution time manageable. Today's QGEN can already quickly generate thousands of queries, and can satisfy the response time demands of a true OLAP workload. To mimic true ad-hoc queries, however, it will be necessary to model iterative, scenario-based query sequences, including changes to the select-list columns, group-by or order-by clauses, as well as function substitutions (min, max, average, etc). These substitutions reduce query-to-query comparability. Preliminary research has confirmed that the increase in queries executed leads to continued stability in the overall metric even with this increased variability.

The next stage of QGEN development will coordinate group-by, order-by and select-list substitutions (using dependency graphs and more complex statistical models such as Hidden Markov Models) to provide the truly random query generation that OLAP benchmarking will require.

## 6. CONCLUSION

In this paper we introduced QGEN, a flexible, high-level query generator optimized for DSS performance evaluation. QGEN is able to generate arbitrary, comparable query sets, which conform to a selected statistical profile without requiring that the queries be statically defined or disclosed prior to testing. It is currently being tested by the TPC for use in its new DSS benchmark TPC-DS. Its close integration with MUDD, a multi dimensional data generator, combined with its elegant query template language ease query workload development and maintenance.

## 7. ACKNOWLEDGMENTS

The development of the data generation prototype that this paper discusses has been funded by the TPC, as part of their on-going efforts to define comparable, relevant and timely decision support benchmarks. They, in turn, rely on the dedication and contribution of the representatives that TPC member companies provide to the TPC's benchmark development subcommittees. The authors would like to thank the TPC, and the members of the TPC-DS subcommittee for their contribution to this effort.

## 8. REFERENCES

[1] Slutz, D. *Massive Stochastic Testing of SQL, Proc*. 24th Int. Conf. Very Large Data Bases, VLDB, 1998

[2] ISO/IEC 9075. *Database Language SQL,* International Standard ISO/IEC 9075:1992, American National Standard X3.135-1992, ANSI, New York, NY 10036, November 1992.

[3] Kimball, R. The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses. John Wiley & Sons, 1996

[4] Poess, M. and Floyd, C., "New TPC Benchmarks for Decision Support and Web Commerce". ACM SIGMOD RECORD, Vol 29, No 4 (Dec 2000)

[5] Poess, M., Smith B., Kollár L., Larson P.: *TPC-DS: Taking Decision Support Benchmarking to the Next Level*. SIGMOD Conference 2002

[6] Stephens, J., Poess, M.: Mudd: A Multi-Dimensional Data Generator, WOSP 2004

[7] Transaction Processing Performance Council (TPC), "TPC Benchmark D (Decision Support)", May 1995 http://www.tpc.org/tpcd/spec/tpcd_current.pdf

[8] Transaction Processing Performance Council (TPC), "TPC-H Specification Version 2.1.0", August 2003 http://www.tpc.org/tpch/spec/tpch2.1.0.pdf

[9] Transaction Processing Performance Council (TPC), "TPC-R Specification Version 2.1.0", August 2003 http://www.tpc.org/tpcr/spec/tpcr_2.1.0.pdf

[10] 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. http://www.census.gov/mrts/www/data/html/nsal01.html