Variations of the Star Schema Benchmark to Test the Effects of Data Skew on Query Performance

Tilmann Rabl
Middleware Systems
Reseach Group
University of Toronto
Ontario, Canada
tilmann@msrg.utoronto.ca

Meikel Poess
Oracle Corporation
Califonia, USA
meikel.poess@oracle.com

Hans-Arno Jacobsen
Middleware Systems
Research Group
University of Toronto
Ontario, Canada
jacobsen@eecg.utoronto.ca

Patrick O'Neil
Department of Math and C.S.
University of Massachusetts
Boston
Massachusetts, USA
poneil@cs.umb.edu

Elizabeth O'Neil
Department of Math and C.S.
University of Massachusetts
Boston
Massachusetts, USA
eoneil@cs.umb.edu

ABSTRACT

The Star Schema Benchmark (SSB), has been widely used to evaluate the performance of database management systems when executing star schema queries. SSB, based on the well known industry standard benchmark TPC-H, shares some of its drawbacks, most notably, its uniform data distributions. Today's systems rely heavily on sophisticated cost-based query optimizers to generate the most efficient query execution plans. A benchmark that evaluates optimizer's capability to generate optimal execution plans under all circumstances must provide the rich data set details on which optimizers rely (uniform and non-uniform distributions, data sparsity, etc.). This is also true for other database system parts, such as indices and operators, and ultimately holds for an end-to-end benchmark as well. SSB's data generator, based on TPC-H's dbgen, is not easy to adapt to different data distributions as its meta data and actual data generation implementations are not separated. In this paper, we motivate the need for a new revision of SSB that includes non-uniform data distributions. We list what specific modifications are required to SSB to implement non-uniform data sets and we demonstrate how to implement these modifications in the Parallel Data Generator Framework to generate both the data and query sets.

Categories and Subject Descriptors

K.6.2 [**Management of Computing and Information Systems**]: Installation Management—*benchmark, performance and usage measurement*

Keywords

Star Schema Benchmark; PDGF; data skew; data generation; TPC-H

1. INTRODUCTION

Traditionally, relational schemas were designed using database normalization techniques pioneered by Edgar F. Codd [4]. Database normalization minimizes data redundancy and data dependencies. However, Kimball argues that this traditional approach to designing schemas is not suitable for decision support systems because of poor query performance and usability [7]. He further argues that the design of a decision support system should follow a star schema, which features a fact table containing real live events, e.g., sales, returns etc., and dimensions further describing these events by supplying attributes such as sales price, item descriptions etc. While traditional normalized schemas are designed to reduce the overhead of updates, star schemas are designed to increase the ease-of-use and speed of retrieval. These fundamental design goals are achieved by minimizing the number of tables that need to be joined by queries.

Having realized the need to measure the performance of decision support systems in the early 1990s, the Transaction Processing Performance Council (TPC) released its first decision support benchmark, TPC-D, in April 1994. Because at that time the majority of systems followed Codd's normalization approach, TPC-D is based on a 3rd normal form schema consisting of 8 tables. Even TPC-H, TPC-D's successor, which was released in April 1999 continued using the same 3rd normal form schema.

While for the technology available at that time, TPC-D and TPC-H imposed many challenges, both on hardware and on database management systems (DBMS). However, their 3rd normal form schema and uniform data distributions are not representative any more of current decision support systems. As a consequence, O'Neil et al. proposed a star schema version of TPC-H, the Star Schema Benchmark (SSB) in [11]. SSB denormalizes the data that TPC-H's data generator, dbgen, generates. However, its underlying

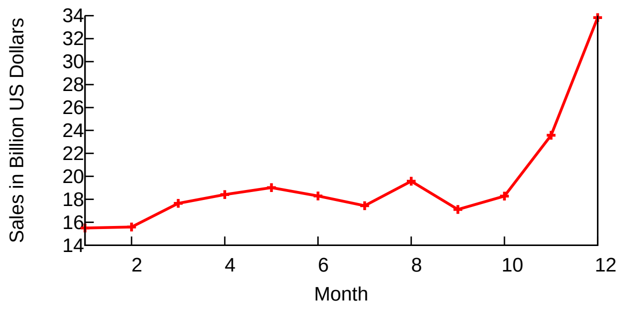

Figure 1: US Retail Sales in Billion US Dollars as Reported by the US Cenus

data generation principles were not changed from TPC-H, especially its uniform data distributions.

We argue that uniform data distributions are not representative for real life systems because of naturally occurring data skew. For instance, retail sales tend to be highly skewed towards the end of the year, customers are clustered in densely populated areas and popular items are sold more often than unpopular items.

Non-uniform data distributions impose great challenges to database management systems. For instance, *data placement algorithms*, especially those on large scale shared nothing systems, are sensitive to data skew, but also common operations such as *join*, *group-by* and *sort* operations can be affected by data skew, most notably the *query optimizer*, is highly challenged to generate the optimal execution plan for queries in the presence of data skew. This is supported in a SIGMOD 1997 presentation by Levine and Stephens in which they noted that a successor of TPC-D should include data skew [2].

Consider the retail sales numbers per month as reported by the US census in Figure 1. The graph shows that US retail sales vary between 15 Billion in January and 35 Billion in December. Sales stay under 20 Billion per month until November when they shoot to 25 Billion and then to 35 Billion in December. It is not uncommon that during the year-end holiday season, often as much as 30% of annual sales are made within the last two months of the year (and much of that in the last half of December) [14].

TPC-DS, published in April 2012, includes some data skew [10], implemented using "comparability zones", which are ranges in the data domain of columns with uniform distributions. The number of comparability zones for a particular column is limited only by the number of unique values in this column. However, because of the underlying design principles of TPC-DS, execution of individual queries is restricted to one comparability zone. While TPC-DS's approach for data skew is a step in the right direction, it limits the effect of data skew greatly, because individual queries still operate on uniform data.

So far there has not been any industry standard benchmark that tests query performance in the presence of highly skewed data. This paper addresses the need for a modern decision support benchmark that includes highly skewed data. The resulting new version of SSB includes data skew in single fact table measures, single dimension hierarchies and multiple dimension hierarchies. To demonstrate the effects of data skew on query performance, several experiments were run on a publicly available DBMS. The results of these experiments underline the need for the introduction of data skew in SSB.

Because SSB's current data generator is an adapted version of TPC-H's data generator, which hard codes most of the data characteristics, a new data generator was implemented using the Parallel Data Generation Framework (PDGF). It's design allows for quickly modifying data distributions as well as making structural changes to the existing tables. In addition it allows the generation of valid SQL from query templates similar to qgen of SSB. This is particularly important as PDGF allows for the generation of queries in the presence of data skew. While this paper demonstrate the key aspects of the data generator, the fully functional data generator is available at `http://www.paralledatageneration.org/`.

In summary, our contributions in this paper are:

- We design and implement a complete data generator for SSB based on the Parallel Data Generation Framework (PDGF) that can be easily adapted and extended.
- We design and implement a query generator for SSB based on PDGF, which enables the generation of more realistic queries with parameters that originate from the data set rather than randomly chosen values.
- We introduce skew into SSB's data model in a consistent way, which makes it possible to reason about the selectivity of queries.
- We implement the proposed data skew in our data generator and demonstrate the effects of the skew on query processing times.

The remainder of this paper is organized as follows. Section 2 reviews those parts of SSB that are relevant to understanding the concepts discussed in this paper. Section 3 presents our data and query generator implementations in PDGF. Section 4 suggests and discusses variations of the star schema benchmark that include various degrees of data skew including experimental results. Before concluding our paper, we discuss related work in Section 5.

2. STAR SCHEMA BENCHMARK

The Star Schema Benchmark is a variation of the well studied TPC-H benchmark [12], which models the data warehouse of a whole sale supplier. TPC-H models the data in 3rd normal form, while SSB implements the same logical data in a traditional star schema, where the central fact table Lineorder contains the "sales transaction information" of the modeled retailer in form of different types of measures as described in [7]. SSB's queries are simplified versions of the queries in TPC-H. They are organized in four flights of three to four queries each.

This paper focuses on data skew in columns that are used for selectivity predicates because data skew in those columns exhibit the largest impact on query performance. Hence, the data set and query set need to be considered when introducing skew into SSB. Therefore, the following two sections first recap the main characteristics of SSB's schema and data set and then analyze the four query flights to ultimately suggest a representative set of columns for which data skew could be implemented.

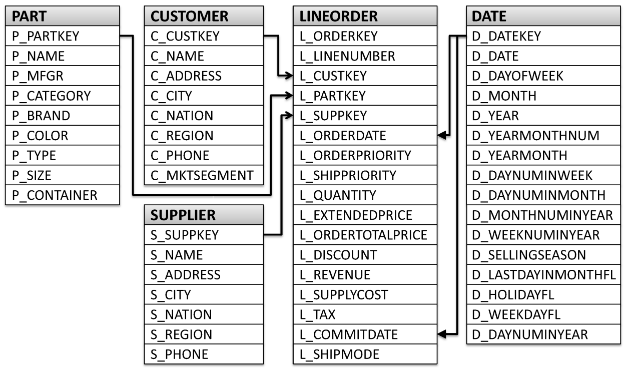

Figure 2: SSB Schema

The uniform characteristics of the TPC-H [1] data set have been studied extensively in [13]. According to the TPC-H specification the term "random" means "independently selected and uniformly distributed over the specified range of values" . That is, n unique values V of a column are uniformly distributed if $P(V = v) = \frac{1}{n}$. Because TPC-H and SSB's data generators use pseudo random number generators and data is generated independently in parallel on multiple computers, perfectly uniform data distributions are impossible to guarantee. Hence, we follow the definition of uniform as presented in [13].

2.1 SSB Schema and Data Population

In order to create a star schema of TPC-H, SSB denormalizes several tables: (i) the Order and Lineitem tables are denormalized into a single Lineorder fact table and (ii) the Nation and Region tables are denormalized into the Customer and Supplier tables and a city column is added. This simplifies the schema considerably, both for writing queries and computing queries as the two largest tables of TPC-H are pre-joined. Queries do not have to perform the join and users writing queries against the schema do not have to express the join in their queries. In SSB, the Partsupp table is removed, because this table has a different temporal granularity than the Lineorder table. This is an inconsistency in the original TPC-D and TPC-H schemas as described in [11]. Finally, an additional Date table is added, which is commonly used in star schemas. To summarize, SSB consists of one large fact table (Lineorder) and four dimensions (Customer, Supplier, Part, Date). The complete schema is depicted in Figure 2.

Like in TPC-H, all data in SSB is uniformly distributed. To be consistent with the general star schema approach the definition of the SSB tables is slightly different from the definition of TPC-H. The most notable change is the introduction of selectivity hierarchies in all dimension tables. They are similar to the manufacturer/brand hierarchy in TPC-H, where the manufacturer (P_Mfgr) value is a prefix of the brand (P_Brand) value. We denote this dependency with P_Brand $\rightarrow$ P_Mfgr, meaning that for each P_Brand value there is exactly one P_Mfgr value. In the Part table, SSB adds a category, which extends the TPC-H hierarchy to P_Brand1 $\rightarrow$ P_Category $\rightarrow$ P_Mfgr. Because all data is uniformly distributed this hierarchy serves as a simple way to control row selectivity of queries. There are five different values for P_Mfgr, resulting in selectivity of $\frac{1}{5}$ each. For each of those there exist five different values in P_Category with a selectivity of $\frac{1}{25}$ each. Finally, the P_Brand1 field has 40 variations per P_Category value for a selectivity of $\frac{1}{1000}$ each. The hierarchies in Customer and Supplier are C_City $\rightarrow$ C_Nation $\rightarrow$ C_Region and S_City $\rightarrow$ S_Nation $\rightarrow$ S_Region, respectively. Date features two natural hierarchies D_DayNumInMonth $\rightarrow$ D_Month $\rightarrow$ D_Year and D_DayNumInWeek $\rightarrow$ D_Week $\rightarrow$ D_Year .

2.2 SSB Queries

Instead of TPC-H's 22 queries, O'Neil et al. propose four flights of queries that each consist of three to four queries with varying selectivity [11]. Each flight consists of a sequence of queries that someone working with data warehouse system would ask, e.g., for a drill down.

Query Flight 1 (Q1.1, Q1.2 and Q1.3) is based on TPC-H's Query 6 (see Listing 1 for the SQL code of Q1.1). It performs a single join between the Lineorder table and the small Date dimension table. Selectivity predicates are set on D_Year, Lo_Discount and Lo_Quantity. Flight 1 simulates a drill down into the Date dimension. From one query of the flight to the next join selectivity is reduced by refining the predicate on the Date component. Q1.1's Date range is one year, Q1.2 selects data within a range of one month and Q1.3's refines this further by selecting data of one week. Q1.3 further changes the predicate on Lo_Quantity from `less than n` to `between n and m`.

```sql
SELECT SUM(lo_extendedprice*lo_discount) AS revenue
  FROM lineorder, date
 WHERE lo_orderdate = d_datekey
   AND d_year = 1993
   AND lo_discount BETWEEN 1 AND 3
   AND lo_quantity < 25;
```

Listing 1: Query Q1.1 of SSB

In order to measure the effect of data skew on query execution times of queries in Flight 1, we propose data skew in (i) Lo_Orderdate, (ii) Lo_Discount and (iii) Lo_Quantity. Data skew in the Lo_Orderdate column is realistic as retail sales are low and flat in the summer, but pick up steeply after Thanksgiving to top in December (see Figure 1). Data Skew in Lo_Discount and Lo_Quantity are also realistic, which can be categorized as single column skew of single tables because these columns are statistically independent.

Queries in Flight 2 join the Lineorder, Date, Part and Supplier tables (see Listing 2 for Q2.1). They aggregate and sort data on Year and Brand. Selectivity is modified using predicates on P_Category, S_Region and P_Brand. That is, they use the P_Brand1 $\rightarrow$ P_Category $\rightarrow$ P_Mfgr hierarchy in the Part table.

```sql
SELECT SUM(lo_revenue), d_year, p_brand1
  FROM lineorder, date, part, supplier
 WHERE lo_orderdate = d_datekey
   AND lo_partkey = p_partkey
   AND lo_suppkey = s_suppkey
   AND p_category = 'MFGR#12'
   AND s_region = 'AMERICA'
 GROUP BY d_year, p_brand1
 ORDER BY d_year, p_brand1;
```

Listing 2: Query Q2.1 of SSB

In addition to data skew in Lo_Orderdate, as proposed for Flight 1, for Flight 2 we propose data skew in P_Category and S_Region, which is realistic as not every manufacturer produces the same number of parts and not all suppliers are distributed uniformly in all regions.

Flight 3 aggregates revenue data by Customer and Supplier nations as well as the years in which orders were placed. Starting from a specific customer, supplier, region and date, Flight 3 drills down into specific cities and years. It uses the City → Nation → Region hierarchies in Customer and Supplier. Listing 3 shows the third query in Flight 3. Only flights 3 and 4 add disjunctive predicates.

```
SELECT c_city, s_city, d_year,
       SUM(lo_revenue) AS revenue
  FROM customer, lineorder, supplier, date
 WHERE lo_custkey = c_custkey
   AND lo_suppkey = s_suppkey
   AND lo_orderdate = d_datekey
   AND (c_city_skewed='City1'
    OR c_city_skewed='City2')
   AND (s_city_skewed='City2'
    OR s_city_skewed='City3')
   AND d_year >= 1992
   AND d_year <= 1997
 GROUP BY c_city, s_city, d_year
 ORDER BY d_year ASC, revenue DESC;
```

Listing 3: Query Q3.3 of SSB

As this flight eventually drills down into one month, data skew in the Lo_Orderdate and the region columns makes sense for this query as well.

Flight 4 is the most complex one, as it joins all tables. Query Q4.1 can be seen in Listing 4. It drills down into region and manufacturer, using both the P_Brand → P_Category → P_Mfgr and the C_City → C_Nation → C_Region hierarchies in Supplier. Hence, data skew in the Customer, Part and Supplier hierarchies makes sense.

```
SELECT d_year, c_nation,
       SUM(lo_revenue - lo_supplycost) AS profit
  FROM date, customer, supplier, part, lineorder
 WHERE lo_custkey = c_custkey
   AND lo_suppkey = s_suppkey
   AND lo_partkey = p_partkey
   AND lo_orderdate = d_datekey
   AND c_region = 'AMERICA'
   AND s_region = 'AMERICA'
   AND (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
 GROUP BY d_year, c_nation
 ORDER BY d_year, c_nation;
```

Listing 4: Query Q4.1 of SSB

The above query analysis shows that implementing data skew in the single columns Lo_Discount and Lo_Quantity, as well as in the dimension hierarchies P_Brand → P_Category → of Part, C_City → C_Nation → C_Region of Customer, and C_City → C_Nation → C_Region of Supplier is sufficient to vary the selectivity of all queries and demonstrate the affects of data skew to query performance.

3. SSB IMPLEMENTATION IN PDGF

The Parallel Data Generation Framework (PDGF) is a generic data generator that was implemented at the University of Passau [15]. It was mainly built to generate relational data for database benchmarking purposes. However, it is able to create any kind of data that can be expressed in a relational model. PDGF exploits the parallelism in *xorshift* random number generators (PRNG) to generate complex dependencies by re-calculating dependent row data (a.k.a. field values) rather than storing them, which would greatly limit the maximum data size and parallelism that can be generated on a given system (for details on xorshift generators see, e.g., [9]). The underlying approach is straight forward: the random number generator is a hash function, which can generate any random number out of a sequence in O(1). With this approach every random number in the sequence of the PRNG can be computed independently. Using the random numbers, generated by the PRNG, arbitrary values are generated using mapping functions, dictionary lookups and such. Quickly finding the right random numbers is possible by using a hierarchical seeding strategy. Furthermore, by using bijective permutations it is possible to generate consistent updates of the data where values may be inserted, changed and deleted [6].

3.1 Data Generation in PDGF

Although pseudo random number generation is in general deterministic, i.e., a repeated execution of the data generator produces the identical results, most data generation approaches do not allow a repeated generation of an individual field value without generating data for all preceding rows. This is due to dependencies within the generation. For example, if the generation of a timestamp in a certain row of a table is dependent on the timestamp in the previous row (e.g., $t_n = t_{n-1}+$ random number of seconds) all previous timestamps have to be computed recursively in order to re-calculate each timestamp. With this approach it is impossible to generate data that references such timestamps without reading the data after its generation. For large data sets and distributed data generation, re-reading data is not feasible. To circumvent these kinds of problems, PDGF uses a repeatable data generation methodology. All field values are generated using functions that map random numbers (input) to row data, the *field value generators*. They have to follow the requirement, that the same input number always creates the same output value and that the value generator is stateless. That is, each field value generator needs to take a seed value as input parameter. If field value generators follow this requirement the challenge of repeatable data generation can be reduced to repeatable random number generation. We achieve this by using the seeding hierarchy depicted in Figure 3. The hierarchy assigns a virtual identifier to each field in the database. The identifier is mapped to a random number by iteratively reseeding random number generators (schema → table → column → update → row). It has to be noted that most of these reseeding operations only have to be done during initialization. Output of the seeding hierarchy is a deterministically seeded random number generator that is used to generate the required input numbers for the generation of the field value. This is especially important if complex field values are generated, such as Markov chain generated pseudo text, which does not consume the same number of random values for each field value.

PDGF can be configured using two XML files: the *Schema Configuration File* describes the underlying relational schema of the data, the *Generation Configuration File* defines

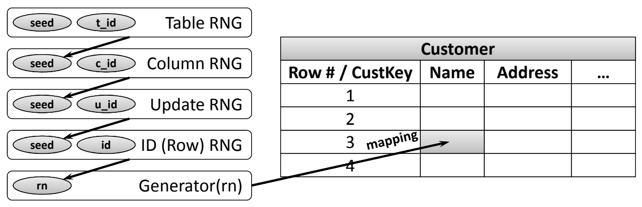

Figure 3: PDGF's Seeding Strategy

how the data is formatted. The schema configuration has a similar structure as the SQL definition of a relational schema, it defines a set of tables which consist of fields. Each field has name, size, data type and generation specification. Listing 5 shows the first two fields of the Supplier table and the definition of dependencies in PDGF. For instance, the S_Name field is defined as the string "Supplier", concatenated with the value of the field S_Suppkey and padded to 9 places. PDGF automatically resolves this dependency if the *OtherFieldValueGenerator* is used. The additional transformations can be specified with meta-generators that change the values generated by subsequent generators. In the example, the padding is done by a *PaddingGenerator* and the prefix is added with another generator.

```
<table name="SUPPLIER">
  <size>${S}</size>

  <field name="S_SUPPKEY" size="" type="NUMERIC"
         primary="true" unique="true">
    <gen_IdGenerator />
  </field>

  <field name="S_NAME" size="25" type="VARCHAR">
    <gen_PrePostfixGenerator>
      <gen_PaddingGenerator>
        <gen_OtherFieldValueGenerator>
          <reference field="S_SUPPKEY" />
        </gen_OtherFieldValueGenerator>
        <character>0</character>
        <padToLeft>true</padToLeft>
        <size>9</size>
      </gen_PaddingGenerator>
      <prefix>Supplier</prefix>
    </gen_PrePostfixGenerator>
  </field>
[..]
```

Listing 5: Excerpt of the Supplier Table Definition

One peculiarity of the SSB schema is the definition of the Lineorder table. It resembles the denormalized Lineitem $\leftrightarrow$ Order relationship of TPC-H. One row in the Order table corresponds to $n \in [1, 7]$ rows in the Lineitem table. In its denormalized version each row of the Lineorder table contains the individual price of the item as well as the total price of the order. Hence, there is a linear dependency between Lineitem and Orders. This is resolved in PDGF in the following way: Instead of treating each line item as a single row we generate all line items of a particular order in one row. This is similar to the generation of Lineorder in the original generator of SSB and the generation of Lineitem and Order in TPC-H, respectively. The PDGF implementation of Lineorder ist shown in Listing 6. The field Lo_Number_Of_Lineitems determines the number of lineitems within a certain order. This field is not printed. All order related values, e.g., Lo_Orderkey, are specified once, while values that are specific to a lineitem, e.g., Lo_Linenumber are specified for each lineitem.

```
    <table name="LINEORDER">
        <size>${L}</size>

        <field name="LO_ORDERKEY" size=""
             type="NUMERIC">
            <gen_FormulaGenerator>
                <formula>
                  (gc.getID() / 8 * 32) +
                      (gc.getID() % 8)
                </formula>
                <decimalPlaces>0</decimalPlaces>
            </gen_FormulaGenerator>
        </field>

        <field name="LO_NUMBER_OF_LINEITEMS" size=""
             type="NUMERIC">
            <gen_LongGenerator>
                <min>1</min>
                <max>7</max>
            </gen_LongGenerator>
        </field>

        <field name="LO_LINENUMBER_1" size=""
             type="NUMERIC">
            <gen_StaticValueGenerator>
                <value>1</value>
            </gen_StaticValueGenerator>
        </field>
[..]
```

Listing 6: Excerpt of the Lineorder Table Definition

The output of Lineorder data as multiple rows is specified in the generation configuration file, which describes the output location and format of the output data. Furthermore, the generation configuration file allows for specifying arbitrary post-processing operations. In Listing 7, an excerpt of the generation specification for Lineorder is shown. The value of the field in Column 2, Lo_Number_Of_Lineitems, determines the number of rows that have to be generated.

```
<table name="LINEORDER" exclude="false">
  <output name="CompiledTemplateOutput">
  <fileTemplate>outputDir + table.getName()
              + fileEnding</fileTemplate>
  <outputDir>output/</outputDir>
  <fileEnding>.tbl</fileEnding>
  <charset>UTF-8</charset>
  <sortByRowID>true</sortByRowID>
  <template><!--
    int noRows
        =(fields[1].getPlainValue()).intValue();
    for (int i = 0; i < noRows; i++) {
      buffer.append( fields[0] ); // LO_ORDERKEY
      buffer.append('|').append( fields[2+i] );
[..]
```

Listing 7: Excerpt of the Generation Definition for Lineorder

Post-processing in the generation configuration, as shown in Listing 7, is only necessary if the relational definition in the schema definition file is not identical to the format on disk.

Another example in the SSB schema is the address information. Because the dimension tables are denormalized the Region and Nation fields have to be generated consistently (e.g., France should always be in Europe). We implemented this by specifying a virtual table Region_Nation, which is referenced by other tables and is not actually generated. This way, we ensure consistency in the data generation because the virtual table Region_Nation contains only valid Region-Nation combinations.

3.2 Query Generation in PDGF

In addition to the data generation, we also designed and implemented a query generator in PDGF, which converts the SSB query templates into valid SQL. In order to generate a large query set, many benchmarks including TPC-H, TPC-DS and SSB define query templates that can be converted into valid SQL queries. For TPC-H and TPC-DS this is done by tools that substitute tagged portions of queries with valid values, such as selectivity predicates, aggregation functions or columns (qgen, DSQgen) [14].In order for such tools to generate a valid and predictable workload they need to have knowledge about the schema and data set. Otherwise the chosen values, e.g., selectivity predicates might not qualify the desired number of rows. TPC-H's qgen and its counterpart for SSB generate queries by selecting random values within the possible ranges of the data. The following shows the definition of the supplier name ranges as specified in the TPC-H specification: `S_NAME text appended with digit ["Supplier", S_SUPPKEY]`

TPC-H's and SSB's qgen implementations have major drawbacks: (i) the parameter ranges for each substitution parameter are hard-coded into a c-program, (ii) scale factor dependent substitution parameter handing is difficult, and (iii) this approach only works on uniformly distributed data.

Instead of hard-coding valid parameter ranges for each query, we use PDGF's post-processing functionality to implement our query generator for SSB. For each query we define one table that is populated with all valid parameters of that query. We refer to these tables as *parameter tables*. The key benefit of our approach is that query parameters are generated from existing values in the data set. Using our approach we generate queries by selecting random field values from the data set, i.e., field values that actually exist in the generated data. This is a more flexible approach as it does not statically define the data range.

In Listing 8, an excerpt for the table specifying the parameters for Query Q1.1 can be seen.

```
<table name="QUERYPARAMETERS_Q1.1">
  <size>13 * ${QUERY_ROUNDS}</size>
  <field name="YEAR" size="4" type="NUMERIC">
    <gen_LongGenerator>
      <min>1993</min>
      <max>1997</max>
    </gen_LongGenerator>
  </field>
[..]
</table>
```

Listing 8: Schema Definition of the Parameter Table for Query Q1.1 (excerpt)

The query template itself is specified in the generation configuration. For Query Q1.1 this can be seen in Listing 9. As can be seen in the listing, the template is embedded in a XML comment in Java-style String format. This format will be compiled into a regular Java binary at runtime. Future versions of PDGF will include a new output plugin that is able to read the template from files and directly generate this Java representation.

```
<table name="QUERY_PARAMETERS" exclude="false" >
  <output name="CompiledTemplateOutput" >
    [..]
    <template><!--
int y = (fields[0].getPlainValue()).intValue();
int d = (fields[1].getPlainValue()).intValue();
int q = (fields[2].getPlainValue()).intValue();
String n = pdgf.util.Constants.DEFAULT_LINESEPARATOR;
buffer.append("-- Q1.1" + n);
buffer.append("select sum(lo_extendedprice *");
buffer.append("        lo_discount) as revenue" + n);
buffer.append("from lineorder, date" + n);
buffer.append("where lo_orderdate = d_datekey" + n);
buffer.append("and d_year = " + y + n);
buffer.append("and lo_disc between " + (d - 1));
buffer.append("                and " + (d + 1) + n);
buffer.append("and lo_quantity < " + q + ";" + n);
 --></template>
  </output>
</table>
```

Listing 9: Excerpt of the Template Specification for Query Q1.1

4. STAR SCHEMA BENCHMARK VARIATIONS

Our PDGF implementation of the original Star Schema Benchmark enables easy alterations to the schema and the data generation as well as the query generation. Columns can be easily added or data distributions changed and queries modified and added. This section discusses different approaches of introducing data skew into the original SSB and the reasoning why some of these changes work and some do not. The variations are based on the findings of Section 2.2 and can be subdivided into four categories:

- Skew in foreign key relations
- Skew in a fact table measures
- Skew in a single dimension hierarchies
- Skew in multiple dimension hierarchies

We discuss each of the proposed data skews below and conduct experiments to show their effects on query elapsed times. Each experiment focuses on one of the above proposed data skews by running queries on the original uniform and on the proposed skewed data sets. For each experiment we choose a representative query template, generate queries for all possible substitution parameters and run them in three modes: (i) Index forced: The query optimizer is forced to use indexes; (ii) Index disabled: The query optimizer is not allowed to use indexes, forcing the system to perform full table scans; and (iii) Index automatic: The query optimizer is free to utilize indexes.

We do not claim that the above three modes are an exhaustive list to analyze the effects of data skew on query elapsed time. However, indexes are widely used to improve query performance and index technology is available on most relational database systems (RDBMS) today. With the three modes, we demonstrate the ability/inability of the query optimizer to choose the best performing plan for each of the substitution parameters and any related side effects, such as large scans and large intermediate result sets. We further demonstrate the difference in query processing time for uniform and skewed data under equal conditions.

Our experiments were conducted on a publicly available RDBMS as a proof of concept. The findings are RDBMS agnostic as the technology used is available in all of today's RDBMS implementations. Thus, the following experiments are not conducting a performance analysis of a specific RDBMS. They are included to show the effects of the proposed distribution changes, thereby underlining the importance of data skew in database benchmarks. The displayed elapsed times are averages from five consecutive runs of the same query.

4.1 Skew in Foreign Key Relations

The characteristics of real life data when loaded into a star schema result in skew of foreign key relations, i.e., fact table foreign keys. For instance, popular items are sold more often than unpopular, some customer purchase more often than others and sales do not occur uniform throughout the year, but tend to be highly skewed towards the holiday season (see also Figure 1).

These characteristics of real life data result in potentially highly skewed foreign key distributions, which impact the way joins are executed, data structures are allocated and plan decisions are made by query optimizers. Implementing skew in foreign keys is easy in PDGF. However, there are drawbacks to adding this kind of skew. Due to surrogate keys being used in star schemas, distributions in foreign keys cannot be correlated to the values in the dimensions that they reference, e.g., values in hierarchy fields. As a result skew in foreign keys does not necessarily translate into skew in the join to dimensions. The high selectivity in dimension fields evens out the skew in the foreign keys.

As an example consider Query Q2.1 in Listing 2. Even if we introduce skew into the foreign keys to the Part table, i.e., Lo_Partkey, many Lineorder tuples will still match the same P_Mfgr value. Since there are only 5 different P_Mfgr values, the resulting selectivities per P_Mfgr value will be almost uniformly distributed. The same is true for the P_Category and P_Brand1 selectivities. For this reason, we do not introduce skew in foreign key references.

In the original SSB the foreign keys, Lo_Custkey, Lo_Partkey, Lo_Suppkey and Lo_Orderdate are distributed uniformly. E.g., the keys in Lo_Orderdate have a coefficient of variation of 0.0044. This allows us to introduce skew into the values of the dimensions themselves. Due to the uniform distributions of the foreign keys the skew of dimension values directly translates into skew of the foreign keys. This is a better way to control the skew in the cardinality of joins between the fact table and dimension tables.

4.2 Skew in Fact Table Measures

Lo_Quantity is an additive measure of the Lineorder fact table. It contains natural numbers $x \in [1, 50]$. In the original uniform distribution values of Lo_Quantity occur with a likelihood of about 0.02 with a coefficient of variation of 0.000284. We modify the Lo_Quantity distribution to skew quantities towards smaller values in the following way: Let $x \in [1, 50]$ be the values for Lo_Quantity, then the likelihood of x is $p(x) = P(X = x) = \frac{0.3}{1.3^x}$. In order to calculate the number of rows with value x, i.e., $r(x)$, we multiply n by the table cardinality, $n = |\text{Lineorder}| : r(x) = n * p(x)$. This results in a exponential probability distribution.

In PDGF this kind of distribution can be implemented very easily by adding a distribution node to the data generator specification. This can be seen in Listing 10. The exponential distribution is parameterized with $\lambda = 0.26235$, which results in the desired distribution.

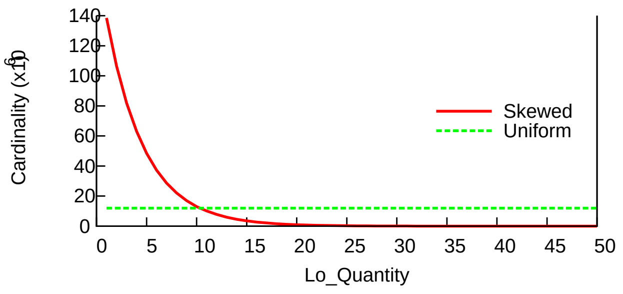

Figure 4: Lo_Quantity Distribution

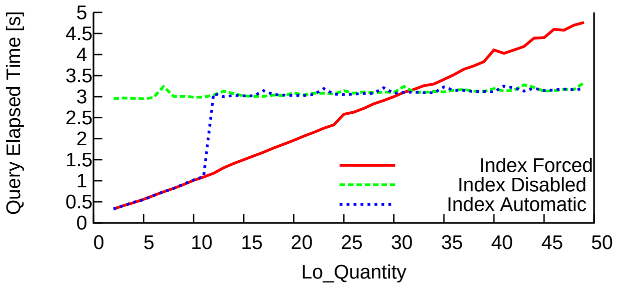

Figure 5: Elapse Times of Query Q1.1 with Uniformly Distributed Lo_Quantity

```
<field name="LO_QUANTITY" size="2" type="NUMERIC">
  <gen_LongGenerator>
    <min>1</min>
    <max>50</max>
    <distribution name="Exponential"
        lambda="0.26235" />
  </gen_LongGenerator>
</field>
```

Listing 10: Schema Definition of Lo_Quantity

Figure 4 shows the uniform (original) and skewed distributions of Lo_Quantity. The x-axis plots the values for Lo_Quantity ranging from 1 to 50 and the y-axis plots the number of rows with those values. The cardinalities in the uniform case for scale factor 100 are constant at around 12 Million with a coefficient of variation of 0.00000557, while the cardinalities in the skewed case start at 140 Million and decrease steeply to 336 with a coefficient of variation of 0.047465541.

Using the skewed and uniform distributions for Lo_Quantity we run Q1.1 and Q1.2 of Flight 1, which contain selectivity predicates on Lo_Quantity. Query Q1.1 contains the "less than" predicate `lo_quantity < x`, while Query Q1.2 contains the "between" predicate `lo_quantity BETWEEN x AND x+9`. We vary the selectivity predicates on Lo_Quantity in the following way: for Query Q1.1 $x \in [2, 51]$ and Query Q1.2 $x \in [1, 41]$. To take advantage of the extreme distribution of Lo_Quantity, we create an index on Lo_Quantity.

Figures 5 and 6 show the elapsed times of Query Q1.1 us-

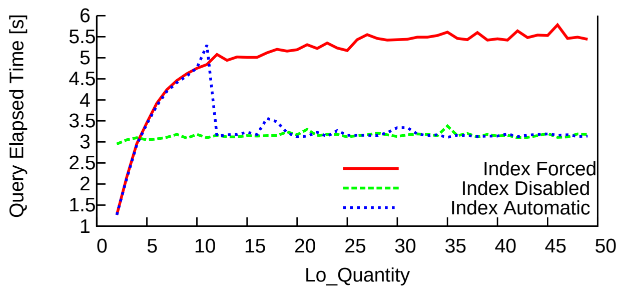

Figure 6: Elapse Times of Query Q1.1 with Skewed Distributed Lo_Quantity

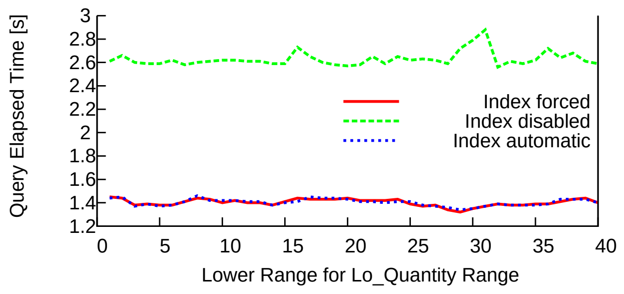

Figure 7: Effect of skewed Lo_Quantity on Query Q1.2 elapsed times

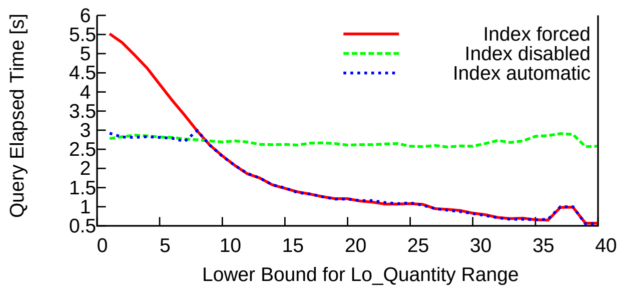

Figure 8: Effect of skewed Lo_Quantity on Query Q1.2 elapsed times

ing the three modes explained above and varying the predicate `lo_quantity < x`, $x \in [2, 51]$ on both the uniform and skewed data sets. The solid line in Figure 5 shows the elapsed times of index driven queries. The elapsed times increase linearly with increased values of Lo_quantity. It starts with about 0.4s at $x = 2$ and increases linearly to 4.8s at $x = 51$. This is not surprising as the advantage of the index diminishes with increased values for x, i.e., the number of rows qualifying for the index driven join increases-linearly with increased values of x. The dashed line shows the elapsed time of the non-index driven join queries. This line is flat at about 3s. This is also not surprising as the non-index driven join queries have to scan the entire Lineorder table regardless of substitution parameters. That is, the index driven queries outperform the non-index driven queries until $x = 31$. The dotted line shows the elapsed time of queries when the query optimizer is free to choose the best execution plan. This line shows that the query optimizer switches from an index driven join to an non-index driven join at $x = 12$. This is suboptimal as the real cutoff of index driven joins is at about $x = 31$. That is the query optimizer switches too early from an index driven join to a non-index driven join.

Figure 6 repeats the same experiments on skewed data. As in the uniform case the elapsed times of the index driven queries, as indicated by the solid line, and the elapsed times of the non-index driven queries, as indicated by the dashed line follow the skewed Lo_Quantity distribution. Due to the extreme skew in Lo_Quantity, the index driven queries outperform the non-index driven queries only for values $x < 5$. The dotted line shows the elapsed time of queries when the query optimizer is free to choose the best execution plan. It shows that the optimizer switches from an index driven plan to a non-index driven plan at the same value for x as in the uniform experiment, namely at $x = 11$. However, this switch is suboptimal. It should have occurred earlier, i.e., at $x = 5$.

These experiments show that the optimizer factors the selectivity of certain predicates in the query planning. However, the fact that the switch in the query plan from index driven to non-index driven occurs at the same predicate parameter for skewed and uniform data suggests that the optimizer does not consider the data distribution for this decision.

Figures 7 and 8 show the elapsed times of Query Q1.2 on uniform and skewed data in the above described three optimizer modes. Similarly, as in the experiments for Query Q1.1, we vary $x \in [1, 41]$ in `lo_quantity between x and x+9`.

Figure 7 shows that the elapsed times of Query Q1.2 stay constant for each of the three modes regardless of the substitution parameter x. On average the index driven queries execute in 2.63s with a coefficient of variation of 0.024, while the non-index driven queries and the queries chosen in the automatic mode execute on average in 1.42s with a coefficient of variation of 0.021. The index driven queries outperform the non-index driven queries by 46% and the lines for index forced and index automatic overlap.

Figure 8 shows that in the skewed case Query Q1.2 benefits from non-index driven joins in cases of high selectivity. The elapsed times of the non-index driven queries are agnostic to the substitution parameter x. With an average of 2.69s at a coefficient of variation of 0.038, which is similar to the uniform case, they outperform the index driven queries for $x < 10$. The index driven queries follow the distribution for Lo_quantity (see Figure 4), i.e., they monotonously decrease with increasing x. For $x >= 10$ the index driven queries outperform the non-index driven queries. The line showing the elapsed times of queries in the automatic mode indicates that the system makes the right choice switching from non-index driven queries to index driven queries at $x = 10$.

4.3 Skew in Single Dimension Hierarchies

Selectivity in many of the original SSB queries is controlled by predicates on hierarchy columns. Each dimension table defines one three-column hierarchy. Each column of a hierarchy defines a different range of values. Column cardinality increases as individual queries of flights

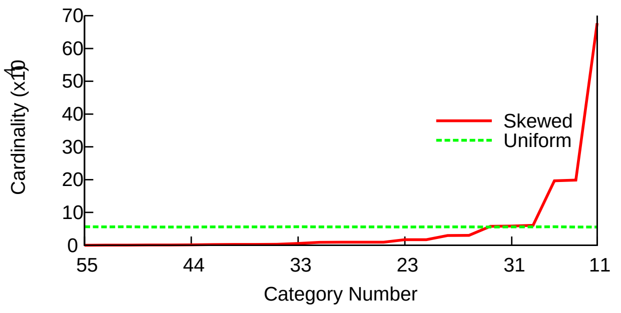

Figure 9: P_Category Distribution Uniform and Skewed

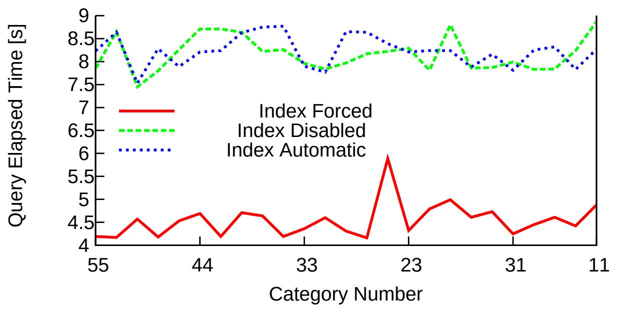

Figure 10: Elapsed Times of Query Q2.1 with Uniformly Distributed P_Category

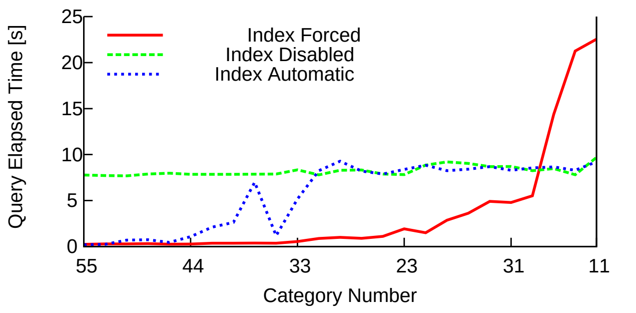

Figure 11: Elapsed Times of Query Q2.1 with Skewed Distributed P_Category

drill down into hierarchies. For example, the Part table defines the following hierarchy: P_Brand → P_Category → P_Mfgr. Values for these hierarchy columns are generated in the following way: Values in P_Mfgr are used as prefixes in P_Category whose values again serve as prefixes in P_Brand. While P_Mfgr contains five different values $m \in \{MFGR\#1, ..., MFGR\#5\}$, P_Category contains 25 different values $c \in \{MFGR\#11, ..., MFGR\#55\}$ and P_Brand contains 1000 values $b \in \{MFGR\#1101, ..., MFGR\#5540\}$. By varying the projection predicates in queries for these three columns, the join selectivity can be varied as well. Consider Query Q2.1 in Listing 2; due to the hierarchy and the uniform distributions in all values and references one can easily see that the selectivity on Part and thus also on Lineorder is $\frac{1}{5}$ of the size of the tables. By changing the restriction to Category, e.g., `p_category = 'MFGR#22'`, the join selectivity can be reduced to $\frac{1}{25}$. The set ratio between the number of rows qualifying a specific substitution parameter is exploited in the query flights.

By introducing skew in these columns the above property may be lost. However, if we control the skew and are able to estimate the frequency of each value, the selectivity calculations are still possible. Therefore, we set a predefined probability for each value in the part hierarchy. For the numbers in Mfgr and Category we use the following probabilities: 1 70%, 2 20%, 3 6%, 4 3%, and 5 1%. For instance, the value “Mfgr#1” appears with a probability of 70% and the category “Mfgr#11” occurs with a probability of $70\% * 70\% = 49\%$. The values where chosen to have one range that has the same probability as in the uniform case (i.e., “MFGR#2” and “MFGR#22”). This makes it possible to specify queries with the same selectivity for the skewed and uniform data. SSB Query Q1.3 includes a range selection on Brand1. Therefore, we define ranges with the same probability in column Brand1. The selectivities in Brand1 are: 1-10 70%, 11-20 25%, 21-30 4.5%, and 31-40 0.5%. Thus brands in the range of “MFRG#2211”-“MFRG#2220” have the same probability as in the uniform case.

Figure 9 displays the uniform and skewed distribution of values in the P_Category column. The cardinalities of the uniform distribution stay at around 4% with a coefficient of variation of 0.0044. The cardinalities in the skewed distribution vary between 0.01% and 48.36% with a coefficient of variation of 2.5028.

Skew in P_Category can be implemented in PDGF as described in Listing 11. It shows the definition of the P_Mfgr field. Each value consists of two parts, the prefix “MFGR#” and a number between one and five. The probability for each number is explicitly defined as specified above. The definition of the fields P_Category and P_Brand1 is implemented in the same way, by explicitly giving the probabilities.

```
<field name="P_MFGR" size="6" type="VARCHAR">
 <gen_PrePostfixGenerator>
  <gen_ProbabilityGenerator>
   <probability value="0.70">
    <gen_StaticValueGenerator>
     <value>1</value>
    </gen_StaticValueGenerator>
   </probability>
   <probability value="0.20">
    <gen_StaticValueGenerator>
     <value>2</value>
    </gen_StaticValueGenerator>
   </probability>
   [..]
  </gen_ProbabilityGenerator>
  <prefix>MFGR#</prefix>
 </gen_PrePostfixGenerator>
</field>
```

Listing 11: Definition of Skew in P_Mfgr Field

Query Q2.1, as defined in Listing 2, can be used to demonstrate the effects of the skewed distribution on query execution times. In Figure 10, the execution times of all possible values of P_Category are shown for the uniform, i.e., original distribution, and in Figure 11, they are shown for the skewed distribution. As can be seen in the graph, the execution times for the queries in the uniform case are almost constant as expected, because the query predicate is an “equal” predicate in Q2.1. Interestingly, the query optimizer consistently chooses the wrong query plan, i.e., an index-driven plan. In the skewed case, the execution times are increasing with the selectivity for the index driven query plan and constant for the plan without index. As in previous experiments, the query optimizer is aware of the skew in the selectivity of P_Category as can be seen at the automatic plan in the skewed case. However, the query optimizer switches from an index to non-index plan too early, thus significantly increasing the query elapsed time. This behavior can only be observed on the skewed data set. With the regular data set the source of the less efficient query plan cannot be determined.

4.4 Skew in Multiple Dimension Hierarchies

This section discusses the introduction of data skew in multiple dimensions and its effect on query elapsed times. Similar to the data skew in the P_Brand1 → P_Category → P_Mfgr hierarchy of the Part table, data skew can be implemented in the City → Nation → Region hierarchies of the Supplier and Customer tables. Having five regions in each table (Supplier, Customer) results in a selectivity of $\frac{1}{5}$ in the uniform case. Each region contains 5 nations, each with a selectivity of $\frac{1}{25}$. Each nation contains 10 cities for a total selectivity of $\frac{1}{250}$. Queries in flights 3 and 4 compare customer with supplier using various levels in the above hierarchies.

For our experiments, we modify S_City and C_City to follow an exponential distribution. For simplicity, we assign each city of the Supplier and Customer tables a unique number: $c_s \in [1, 250]$ for Supplier and $c_c \in [1, 250]$ for Customer. The likelihoods of Supplier and Customer cities are then defined as $p(c_s) = P(C = c_s) = \frac{0.0309}{1.0309^{c_s}}$ and $p(c_c) = P(C = c_c) = \frac{0.04}{1.04^{c_c}}$. In order to calculate the number of rows $r(c_s)$ with value c_s and the number of rows $r(c_c)$ with values c_c, we multiply the likelihoods by the table cardinality $n_s = |\text{Supplier}| : r(c_s) = n_s * p(c_s)$ and $n_c = |\text{Customer}| : r(c_c) = n_c * p(c_c)$. This results in exponential probability distributions for Supplier and Customer cities as depicted in Figure 12. The cardinality of the Customer table is 15 times larger than the cardinality of the Supplier table. In order to plot graphs for S_City and C_City in one graph, we normalized both by dividing the cardinality of each city by the cardinality of the largest city: $max|\text{Supplier}_{\text{City}}| = 6799$, $max|\text{Customer}_{\text{City}}| = 121135$. As depicted in Figure 12 both distributions are similar.

Query Q3.3 (see Listing 3) contains selectivity predicates on Date(D_Year), Supplier(S_City) and Customer(C_City). The predicates on S_City and C_City are disjunctions. We use the same city for each part of the disjunction. Hence, we have two substitution parameters (city1, city2) for Q3.3, where city1 is used as a projection of Supplier and city2 is used as a projection of Customer. Because there is no correlation between the city distributions in Supplier and Customer and the Date table, not all city pairs are guaranteed to occur for every year. Hence, before running our experiments we compute the valid Supplier and Customer city pairs. It turns out that out of the total combination of $250^2 = 62500$ city pairs only 1569 occur within `D_Year=1997` (default for Query Q3.3) in the original SSB data. To reduce the number of experiments we select a subset of 156, i.e., every 10th, city pair. Figure 13 shows the join cardinalities $|C_{c\_city=city1} \bowtie L \bowtie S_{s\_city=city2} \bowtie D_{d\_year=1997}|$ of the subset of valid Supplier and Customer city pairs that we use in subsequent experiments. The graph shows that

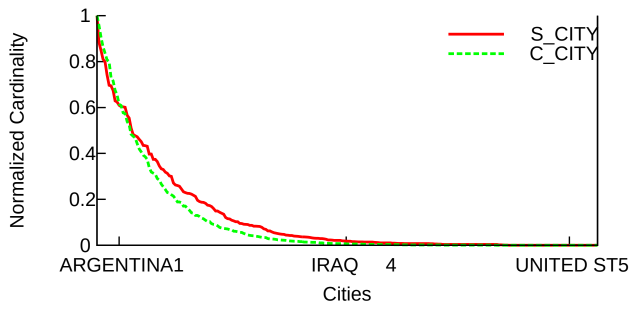

Figure 12: C_City and S_City Skewed Distributions

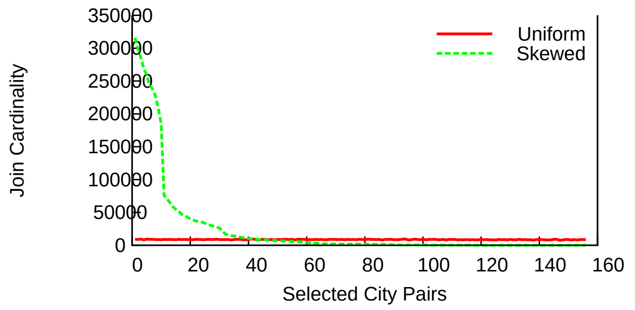

Figure 13: Query Q3.3 Join Cardinalities of the Skewed Distributions

the skewed distributions of cities in Supplier and Customer translate into a similar distribution of the join cardinalities. They start very high at around 300k and decrease negative exponentially to single digit values for high city pairs. At city pair 41 the two lines cross, i.e., before city pair 41 the join cardinalities of the skewed case are much higher compared to the uniform case. After that skewed cardinalities are much lower compared to the uniform cardinalities.

In PDGF the above distribution can be implemented very easily by adding a distribution node to the data generator specification. For the city fields the exponential distribution is implemented as in the Lo_Quantity field as shown above in Listing 10. In this experiment, we are mainly interested in the effect of multiple skewed dimensions and, therefore, we only skew the most selective attribute, which is S_City and C_City, respectively. As discussed in Section 3.1, in our PDGF implementation nation and region are references to a virtual table, as is the nation prefix of the City attribute. The definition of the S_City attribute can be can be seen in Listing 12. A value of S_City is made from a prefix that is a padded or truncated version of a Nation value (strings longer than 9 characters are truncated, shorter strings are padded) and a number between 0 and 9. In order to generate a skewed output, the reference to Nation_Region can be skewed, as can the generation of the number that is concatenated with the prefix. In our PDGF implementation for Supplier we only insert a skew in the reference to Nation_Region, while for Customer, we additionally add a skew in the number generation.

Figures 14 and 15 show the elapsed times of Q3.3 on uniform and skewed distributions of S_City and C_City. Query Q3.3 is more complicated compared to the queries used in previous experiments as it joins four tables: Lineorder, Sup-

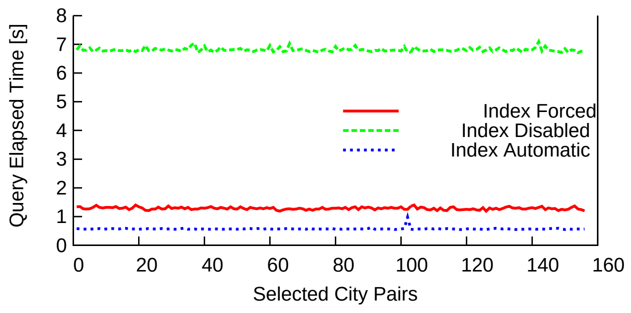

Figure 14: Elapsed Times of Query Q3.3 with Uniform C_City and S_City

```
<field name="S_CITY" size="10" type="VARCHAR">
  <gen_SequentialGenerator concatenateResults="true">
    <gen_PaddingGenerator>
      <gen_DefaultReferenceGenerator id="S_CITY_id">
        <reference table="NATION-REGION"
             field="NATION" />
      </gen_DefaultReferenceGenerator>
      <character> </character>
      <padToLeft>false</padToLeft>
      <size>9</size>
    </gen_PaddingGenerator>
    <gen_LongGenerator>
      <min>0</min>
      <max>9</max>
    </gen_LongGenerator>
  </gen_SequentialGenerator>
</field>
```

Listing 12: Definition of S_City in the Supplier table

plier, Customer and Date. Due to the increased complexity, the choices for indexes and query plans for Query Q3.3. are much larger compared to those in previous experiments. Hence, the index driven join forces the system to use a query plan that does not necessarily coincides with the query plan chosen by the optimizer in automatic mode.

Figure 14 shows the elapsed times for all three modes on a uniform data set. Each line shows a constant query elapsed time regardless of the city pair chosen. This is not surprising as the cardinalities of at which each city pair occurs in the database is constant (see Figure 12. The non-index driven queries have the longest elapsed times while the automatic queries have the shortest elapsed times. The index forced queries execute slightly longer than the index driven queries.

Figure 15 shows the elapsed times for all three modes on the skewed data set. The graph reveals a couple of very interesting characteristics of the system. The elapsed times of the non-index driven queries, dashed line, is more or less constant at about 7s. Queries using the first 20 city pairs, which result in very high join cardinalities (50k to 300k) show slightly higher elapsed times (up to 9s). This is due to the larger intermediate result set at these high join cardinalities. Contrary, the lines for index forced and index automatic plans show a steep decrease in elapsed time starting at city pair 30 for the automatic plan and 34 for the index forced plan. The steep decrease is due to the steep decrease in the join cardinality. That is the index driven queries are able to take advantage of the index to filter out most of the rows to beat the non-index driven queries. Again, this is due to a sub-optimal query plan that was forced in the

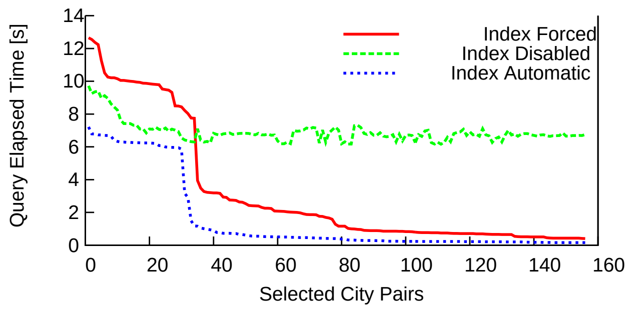

Figure 15: Elapsed Times of Query Q3.3 with Skewed C_City and S_City

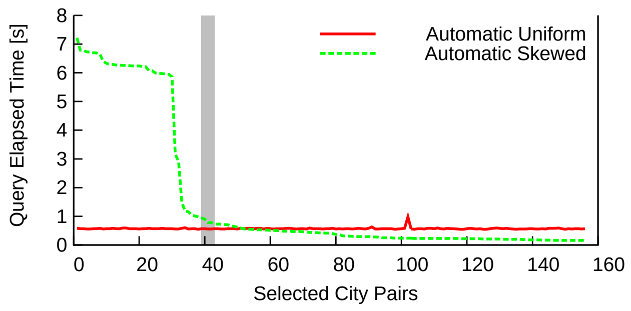

Figure 16: Elapsed Times of Query Q3.3 with Uniform and Skewed C_City and S_City using Automatic Plans

index-forced mode. Also, like in the uniform case, the automatic query outperforms both the index driven and non-index driven queries for all city pairs.

Figure 16 compares the automatic case for the uniform and skewed distributions. The queries run on uniform data execute constantly under 1s, while the queries on skewed data follow the cardinality distribution as described above. With a decrease in join cardinality the queries on skewed data outperform those on uniform data, which is not surprising. However, the cross over point seems to be at city pair 52. We would have expected the cross over to be closer to city pair 41, where the join cardinalities of the skewed and uniform data sets are identical. This shows that the system has a consistently increased latency for skewed data. This trend also holds for the index-forced and no-index plans.

5. RELATED WORK

While the importance of data skew in database processing has been identified and, to some degree, quantified in various publications, especially in the field of parallel systems [8, 16], there is no industry standard benchmark that measures the effect of data skew on query processing. In many cases, performance of parallel DBMS is adversely affected by data skew, because it introduces load imbalances, stresses the creation of internal data structures, and it reveals poor design decisions in algorithms and query optimizers. Having realized the above, Crolotte and Ghazal proposed the introduction of data skew into TPC-H similarly to TPC-DS, i.e., using comparability zones [5]. As a consequence, queries are still run with substitution parameters that are chosen from within a comparability zone with uniform distribution.

[5] discusses two approached how skew can be introduced into the nation keys of the Customer and Supplier tables. The first approach scales the numbers of customers and suppliers who reside in a nation proportional to the actual population of this nation (e.g., obtained from the US census). Although being realistic this approach is not feasible in the context of TPC-H as the queries would impose different workloads for different customers. Their second approach, which is feasible for TPC-H is creating skew with a step function. The first 13 nations have a small population, while the remaining 12 nations have a large population. They implement the above described skew using Teradata SQL syntax with a random function, i.e., they read the original TPC-H data and apply a random function for nation keys.

Closer to our approach for introducing data skew in standard benchmarks is Chaudhuri and Narayasa modified version of TPC-D [3]. TPC-D was the precursor of TPC-H. The two benchmarks share the same schema and data model, but vary in their workload slightly. Hence, the proposed changes also apply to TPC-H. Chaudhuri and Narayasa modified TPC-D/H's data generator dbgen to support Zipfian distributed data in all columns. The parameter to the distribution (ρ), which controls the degree of skew in the data, is given as a parameter to the modified version of dbgen. ρ can be set to any value larger or equal to 0; $\rho = 0$ generates a uniform data whereas $\rho = 4$ generates a highly-skewed distribution. In addition they also implemented functionality that allows dbgen to randomly chose ρ values for each column, thereby creating a "mixed" data distribution.

Their approach differs from Crolotte and Ghazal's approach as it does not guarantee workload predictability, which is a major factor in industry standard benchmarks and in particular in TPC-H. However, Chaudhuri and Narayasa approach can be easier implemented as the data generator is freely available, but it is limited to the Zipf distribution and cannot be customized on a column per column basis.

Data skew has been addressed to some extend in TPC-DS. The TPC after realizing that the uniform data distributions in TPC-D, TPC-H and TPC-R were not challenging today's DBMS vouched to implement data skew in its next decision support benchmark TPC-DS. The TPC opted for a solution that introduced zones of comparability – essentially flat spots in the data distribution – that can be used to provide both the variability and the comparability that the eventual user of the generated data requires. These comparability zones differ in size within the column domain and in number between column domains.

6. CONCLUSION

In this paper, we presented an extension of the Star Schema Benchmark that introduces skew in single table columns and single table hierarchies. Due to the limited extensibility of the original data generator, we implemented a new data generator and query generator based on the Parallel Data Generation Framework. Our extensive experimental analysis shows that the introduction of skew in the data sets can uncover unpredicted behavior in query processing.

PDGF is continuously extended and improved. A recent extension was presented in [6]. It allows for the efficient generation of consistent update data. This enables PDGF to generate more realistic updates on the data compared to TPC-H and SSB. However, we did not explore this option in the presented work. Information on the current status of PDGF as well as downloadable versions can be found on `www.paralleldatageneration.org`.

For future work, we will explore the possibilities of using references to the generated data set in query generation. This will make it possible to introduce skew in the query workload and thus make it possible to examine the influence of query caching and other optimizations in a meaningful way. In combination with the deterministic data generation the reference-based query generation will enable precomputation of query results for certain kinds of queries and thus provide means for verification beyond the capabilities of current benchmarks. We will combine these and the above presented parts into a complete benchmark suite that will make it possible to test the different influences of skew in separated and combined workloads.

7. REFERENCES

[1] TPC Benchmark H. `http://www.tpc.org/tpch/spec/tpch2.15.0.pdf`, 2012.

[2] J. S. Charles Levine. Standard Benchmarks for Database Systems. `http://www.tpc.org/information/sessions/sigmod/sigmod97.ppt`, 1997.

[3] S. Chaudhuri and V. Narasayya. Program for Generating Skewed Data Distributions for TPC-D. `ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew`, 1997.

[4] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(1):377–387, 1970.

[5] A. Crolotte and A. Ghazal. Introducing Skew into the TPC-H Benchmark. In *TPCTC '11*, pages 137–145, 2011.

[6] M. Frank, M. Poess, and T. Rabl. Efficient Update Data Generation for DBMS Benchmark. In *ICPE '12*, 2012.

[7] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley and Sons, Inc., 2002.

[8] M. S. Lakshmi and P. S. Yu. Effect of Skew on Join Performance in Parallel Architectures. In *DPDS*, pages 107–120, 1988.

[9] G. Marsaglia. Xorshift RNGs. *Journal Of Statistical Software*, 8(14):1–6, 2003.

[10] R. O. Nambiar and M. Poess. The Making of TPC-DS. In *VLDB '06*, pages 1049–1058, 2006.

[11] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC '09*, pages 237–252, 2009.

[12] M. Poess and C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Record*, 29(2000):64–71, 2000.

[13] M. Poess, T. Rabl, M. Frank, and M. Danisch. A PDGF Implementation for TPC-H. In *TPCTC*, pages 196–212, 2011.

[14] M. Poess and J. M. Stephens. Generating Thousand Benchmark Queries in Seconds. In *VLDB*, pages 1045–1053, 2004.

[15] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A Data Generator for Cloud-Scale Benchmarking. In *TPCTC '10*, pages 41–56, 2010.

[16] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *VLDB '91*, pages 537–548. Morgan Kaufmann, 1991.