A PDGF Implementation for TPC-H

Meikel Poess [1], Tilmann Rabl[2], Michael Frank[3], Manuel Danisch[3]

[1] Oracle Corporation, 500 Oracle Pkwy, Redwood Shores, CA-94065
meikel.poess@oracle.com
[2] Middleware Systems Research Group, Department of Computer Science ,
University of Toronto, 10 King's College Road, Toronto, Ontario, Canada, M5S 3G4
tilmann.rabl@utoronto.ca
[3] Fakultät für Informatik und Mathematik, Universität Passau, Innstraße 43, 94032 Passau
{frank,danisch}@fim.uni-passau.de

Abstract: With 182 benchmark results[1] from 20 hardware vendors, TPC-H has established itself as the industry standard benchmark to measure performance of decision support systems. The release of TPC-H twelve years ago by the Transaction Processing Performance Council's (TPC) was based on an earlier decision support benchmark, called TPC-D, which was released 1994. TPC-H inherited TPC-D's data and query generators, DBgen and Qgen. As systems evolved over time, maintenance of these tools has become a major burden for the TPC. DBgen and Qgen need to be ported on new hardware architectures and adapted as the system grew in size to multiple terabytes. In this paper we demonstrate how Parallel Data Generation Framework (PDGF), a generic data generator, developed at the University of Passau for massively parallel data generation, can be adapted for TPC-H.

Keywords: Performance Analysis, Benchmark Standards, TPC-H, Data Generation

1 Introduction

Since its introduction in 1999 by the Transaction Processing Performance Council (TPC) 20 system vendors have published 182 benchmark results on hundreds of system configurations using the TPC-H benchmark specification. This establishes TPC-H as the de facto industry standard to measure performance of decision support systems. Closely tight to its specification are its data and query generators, DBgen and Qgen respectively, which are implemented in the programming language C. Their development, originally used in TPC's first decision support benchmarks (TPC-D), was completed in 1994. Since that time, the code has been ported to 20 separate platforms, spanning OS versions from UNIX to Windows, and from VMS, to MVS, to Linux [**Error! Reference source not found.**].

Since the introduction of DBgen and Qgen in 1994 systems used in TPC benchmark publications have evolved greatly causing the maintenance of these tools to be a major burden for the TPC. While systems that published TPC-D benchmarks only

[1] As of May 9th, 2011

employed few single core processors on data warehouse of up to one Terabyte, systems running TPC-H today employ clusters of multi-core processor nodes, totaling hundreds of cores, on data warehouses of up to 30 Terabytes and multi Terabytes of main memory. Recently the frequency at which bugs are reported increased dramatically, which lead to a discussion of completely rewriting DBgen. However, this turned out to be cost prohibitive. As an alternative this paper investigates the feasibility of using the Parallel Data Generation Framework (PDGF), developed at the University of Passau, for TPC-H. Originally developed for massively parallel data generation of cloud scale databases, PDGF has many advantages over DBgen: It is written in the platform independent language Java, which makes portability needless. Studies have shown that it is able to generate terabytes of data quickly and reliably [2]. Its separation into a data generation engine and a file defining the metadata about the data to be generated makes it easily maintainable and, if necessary quickly extensible. Finally, since it is a generic data generation tool, it can also be adapted by other benchmarks in which case the TPC only needs to maintain one data generator.

The remainder of this paper is organized as follows. Section 2 gives a quick overview of the different data generation requirements of TPC-H. Section 3 introduces PDGF and develops the metadata file that allows PDGF to generate TPC-H data. It also explains some of the modifications that needed to be implemented in PDGF to allow for the different data types. In Section 4 a detailed analysis of the data generated by PDGF is presented. The paper concludes in Section 5. We have included the two metadata files (Appendix A) and all SQL compliance queries (Appendix B) as supporting material to this paper.

2 Overview Data Generation in TPC-H

TPC-H models the activity of any industry which manages, sells, and distributes products worldwide. It uses a 3rd normal form schema consisting of eight base tables, (see Figure 1). They are populated with synthetic data, scaled to a scale factor (SF) that determines the size of the raw data outside the database, e.g. SF=1000 means that the sum of all base tables equals 1 Terabyte. Sizes of all tables, except for nation and region scale linearly with SF (see [1,6] for more details on TPC-H).

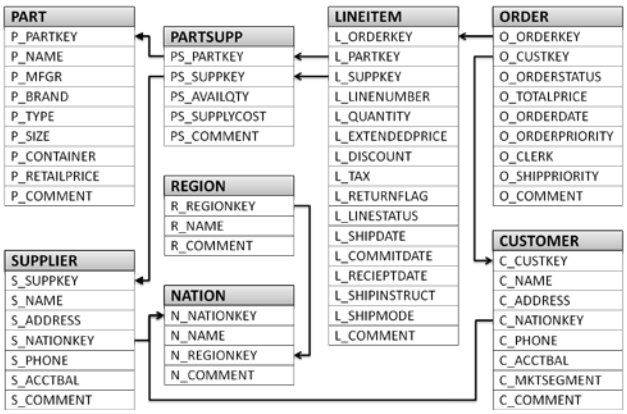

Figure 1: TPC-H Schema

In order to guarantee that every database publication uses the same data in the base tables, the TPC-H specification defines the content of every

column very precisely using the following primitives: Date, Phone Number, Random String, Random Value, Random v-String, Text Appended Digit and Text String. Table 1 shows how these data generation primitives are used in defining column content. For space reasons we only list a representative subset of all TPC-H columns. For a full list of column definition see Clause 4.2 in [6]. In TPC-H the term "random" means independently selected and uniformly distributed over the specified range of values.

Table 1: Example usage of data generation primitives in TPC-H

Column	Use of data generation primitive	Sample output
O_Orderdate	Date, uniformly distributed between 1992-01-01and 1998-08-02	1995-05-26
S_Phone	Phone Number	16-421-927-9442
L_Shipinstruct	Random String [instructions], where Instructions={DELIVER IN PERSON, COLLECT COD,NONE, TAKE BACK RETURN}	TAKE BACK RETURN
S_Nationkey	Random Value [0 .. 24]	23
S_Address	Random v-String[10,40]	vs50U4?e5i
S_Name	Text Appended with Digit ["Supplier", S_Suppkey]	Supplier5628
PS_Comment	Text String [49,198]	dependencies beyo

3 Implementing TPC-H in PDGF

PDGF is an extensible Parallel Data Generation Framework, developed at the University of Passau, to generate Exabytes of synthetic data by utilizing deterministic parallel pseudo random number generators. In its current form it is limited to generating data for relational database management systems (RDBMS). However, its design can be extended to allow for the generation of structurally different data, e.g. XML.

PDGF's architecture is designed for large data sets, maximum performance and easy extensibility. Figure 2 shows a sample setup with three nodes. The controller, executed in the center node, reads meta-data about the schema, its distributions, output format and system configuration from XML files and initiates the data generations by spanning multiple threads (by default one for each core) on each node. The scheduler divides the work and assigns equal sized, continuous portions of the data to each thread. The actual data generation is done by so called

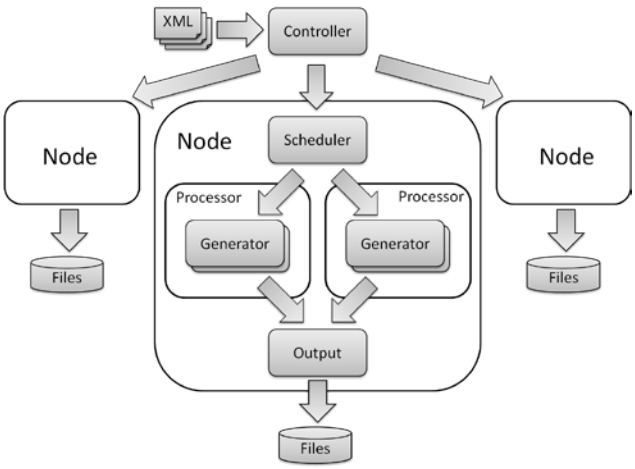

Figure 2: PDGF Architecture

generators, which are executed in threads. To generate non-uniform data the system features various distributions that can be applied to the random numbers.

Its unique seeding approach allows PDGF to generate random values for each field deterministically. To generate a single value for a column (e.g. name), a hierarchy of random number generators is used: Table→Column→Row →NameGenerator. Even for large relational schemas the total number of seeds required can be cached in PDGF. This approach enables PDGF to generate all values for all columns of all tables independently and deterministically. Dependencies of columns, i.e. Intra-Row (e.g. ZIP→city), Intra-Table (e.g. surrogate key sequence) and Inter-Table (e.g. referential integrity) can be resolved without caching all values or re-reading previously generated data back in. For a discussion of the generation of data dependencies please refer to [3].

In [2] a comparison of the generation speed of DBgen vs. PDGF was presented. Although PDGF is a generic data generator, it has a comparable generation speed to DBgen. PDGF includes a range of generators that allow the generation of all common relational data types, these include numeric values, random strings and datestamps. New generators can be added to PDGF as plug-ins. This is especially useful for benchmarks like TPC-H that have very special requirements for its data specification. For example, TPC-H has various interdependencies in the data definition and special data generation rules. Therefore, a plug-in for TPC-H was implemented that encapsulates all TPC-H specific generators. An actual extension to the PDGF core was a cache for the active row. Although in principle all values can be computed it is much more efficient to cache a single row than to compute all values several times for a single row. This is necessary for intra-row dependencies which can be found for example in Part, where P_Retailprice is calculated based on P_Partkey. Besides this extension only TPC-H specific generators had to be implemented. The following paragraphs give details on the implementation and how to configure PDGF to generate data for the generation primitives, presented in Section 2. The generation specification resembles the relational schema: it is an XML file that contains an element table for each table in the schema. Each of the table elements has multiple field subelements. These represent a column in the table. For each table element the PDGF will generate a single file in which each row consists a number of fields that is defined by the field subelements.

Appendix A contains a full list of the TPCH.pdgf file that generates the entire data set for TPC-H for a given SF.

Date (min,max): Since date is a fairly common data type in relational data, PDGF comes with a generic date generator, called DateGenerator. Figure 3 shows how DateGenerator can be configured to generate O_Orderdate, which are uniformly distributed between 1/1/1992 and 8/2/1998 (see StartDate and EndDate tags in the example). To do so, it converts the assigned date range in milliseconds and scales down the random number to the given date range in milliseconds. There are several other fields that require dates. They directly depend

```
<field name="O_ORDERDATE">
  <type>java.sql.Types.DATE</type>
  <generator name="DateGenerator">
    <startDate>1992-01-01</startDate>
    <endDate>1998-08-02</endDate>
  </generator>
</field>
```

Figure 3: Configuration of the DateGenerator for O_Orderdate of Order

on other fields. For example L_Shipdate is defined as a date 1 to 121 days after O_Orderdate. Similar dependencies are defined for the fields L_Receiptdate and L_Commitdate. These intra-row dependencies require special generators that, on the fly, look up dates and compute other dates. In Java, these are implemented as subclasses of the date generator adding simple date arithmetic to implement the dependency. For faster processing they make use of the row cache and the reference lookup in PDGF.

Phone Number: In TPC-H a phone number is defined as a string constructed of four random numbers that are separated by dashes, e.g. 1-650-633-8000. The PhoneNumberGenerator, is specifically designed for TPC-H. For each of the four segments of the phone number a separate random number is generated in the specified interval and the numbers are concatenated. Since there are no further restrictions, the call to PhoneNumberGenerator has no arguments as can be seen for the configuration of S_Phone of the supplier table in Figure 4.

```
<field name="S_PHONE">
  <type>java.sql.Types.VARCHAR</type>
  <size>15</size>
  <generator name="PhoneNumberGenerator" />
</field>
```

Figure 4: Configuration of the PhoneNumberGenerator for S_Phone of Supplier

Random String: Random String values are generated by randomly picking one element from one or multiple lists. If a single list is used PDGF's DictListGenerator can be used. Figure 5 shows the configuration of the O_Orderpriority field, which can be one of {1-URGENT, 2-HIGH, 3-MEDIUM, 4-NOT SPECIFIED, 5-LOW}. In general it randomly chooses values from a dictionary with a uniform distribution. Other distributions can be specified explicitly. The dictionary is stored in a file, whose name can be specified with the file tag. Fields that require multiple lists can be generated with the same generator by creating a dictionary file that contains all combinations of the lists. This is a feasible solution since all TPC-H lists contain only few elements. The maximum number of entries is 150 (see P_Type of the Part table in [6]). For a faster generation PDGF caches dictionaries in memory.

```
<field name="O_ORDERPRIORITY">
  <type>java.sql.Types.VARCHAR</type>
  <size>15</size>
  <generator name="DictList">
    <file>dicts/priorities.dict</file>
  </generator>
</field>
```

Figure 5: Configuration of the DictList generator for O_Orderpriority of Order

Random Value: Random values can be generated using the IntGenerator. It is called with min and max values and an optional distribution function, such as normal, Gaussian or Zipf. TPC-H only requires uniform distributions. Figure 6 shows the configuration of the S_Nationkey. It is randomly picked between 0 and 24.

```
<field name="S_NATIONKEY">
  <type>java.sql.Types.INTEGER</type>
  <generator name="IntGenerator">
    <min>0</min>
    <max>24</max>
  </generator>
</field>
```

Figure 6: Configuration of the random integer number generator of N_Nationkey of Nation

```
<field name="S_ADDRESS">
  <type>java.sql.Types.VARCHAR</type>
  <size>40</size>
  <generator name="RandomVSTring" />
</field>
```

Figure 7: Configuration of the RandomVString generator

Random v-String: In TPC-H Random V-string primitive denotes a randomly generated string over an alphabet of 64 characters. The length of the string is uniformly distributed between a 10 and 40. Random v-String is implemented in PDGF with the RandomVString. It randomly chooses the length of the string between the given min and max values and then fills the string by randomly picking elements from an alphabet of 64 characters. In TPC-H Random v-Strings are used for address fields, such as S_Address. The specification for S_Address is depicted in Figure 7.

Text Appended Digit: In TPC-H the Text Append Digit primitive specifies a field that consists of a text followed by '#' and a random integer number. Most fields use this type in connection with intra-row dependencies. For example, the name of a customer C_Name consists of the text "Customer", a '#' sign and value of the field C_Custkey of the same row. The specification of C_Name can be seen in Figure 8; the digitSource element specifies that the value of C_Custkey will be used as the number in the generation of C_Name. Another special case is the generation of P_Brand, the brand of a part. It dependens on P_Mfgr, the manufacturer of a part. Both fields are text, appended with digits, but the random number of P_Brand is preceded by the random number of P_Mfgr of the same row. Since P_Mfgr is not a number, the generator cannot simply use the digitSource element. To reduce the computational overhead, this is implemented as a special case in the generator. The generator caches the last random number of P_Mfgr in order to reuse it in P_Brand.

```
<field name="C_NAME">
  <type>java.sql.Types.VARCHAR</type>
  <size>25</size>
  <generator name="TextAppendedWithDigit">
    <text>Customer</text>
    <digitSource>C_CUSTKEY</digitSource>
  </generator> </field>
```

Figure 8: Configuration of the TextAppendedWithDigit generator of S_Address of Supplier

Text String: The most difficult primitive in TPC-H is the Text String primitive. It is used in multiple comment fields such as C_Comment of Customer. The generated value is a random substring of a 300 MByte pseudo text file. The length of the string is randomly chosen between specified upper and lower bounds. The offset of the string is also randomly chosen. The 300 MByte file is populated with a grammar definition. The grammar emulates the composition of English texts. TPC-H also specifies a lists of verbs, nouns, adjectives and the like which are used as terminals for the grammar.

```
<field name="N_COMMENT">
  <type>java.sql.Types.VARCHAR</type>
  <size>117</size>
  <generator name="TextString">
    <min>29</min>
    <max>116</max>
  </generator>
</field>
```

Figure 9: Configuration of the TextString generator of N_Comment of Nation

Although it would be possible to cache the pseudo text it can also be computed on the fly. The TextString generator loads word lists and generates sentences using the specified grammar. For performance reasons the text generator is implemented as a singleton object. The XML specification for the TextString generator can be seen in Figure 9. A special case of the TextString primitive is used in field S_Comment of Supplier. 0.05 percent of S_Comment entries are complaints and 0.05 percent of the entries are recommendations. These have to include the string "Customer" followed by a random number of characters, followed by either the string "Complaints" or the string "Recommends". In PDGF text for S_Comment is generated using the same pseudo text generator as above, additionally complaints and recommendations are inserted in the text with given probabilities.

Apart from these types and their specializations, several custom generators for single fields were implemented. These usually have dependencies that make a generic implementation inefficient. An example is L_Extendedprice; it is calculated as L_Quantity * P_Retailprice, where P_Partkey = L_Partkey in the according rows. Obviously, it is easier implement the logic in a generator instead of implementing a generic generator that allows these kinds of dependencies.

4 TPC-H.pdgf Verification

This section describes our approach to verify whether PDGF, using the attached TPC-H.pdgf file generates data that complies with the current TPC-H specification, Version 2.14.0. To demonstrate functional compliance with the current TPC-H specification, we need to analyze whether all columns of all tables contain data that is compliant with Clause 4.2.3 of the specification, which we reviewed in Section 2. First we verify the cardinalities in each table, followed by one section for each data primitives: Date, Phone Number, Random String, Random Value, Random v-String, Text Appended Digit, Text String and Unique Value. At the end of these sections, we list some columns that do not quite follow the generation primitives. They are in a section labeled special cases.

Most of the primitives refer to the term random. 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 $P(V = v) = \frac{1}{n}$. Since we use pseudo random number generators, perfectly uniform distributions are impossible to guarantee. Hence, we define a column C with n unique values V to be uniformly distributed if the coefficient of variation of its values is less than ε. Formally, given the mean of V as $\mu = \frac{1}{n}\sum_{i=1}^{n} v_i$and its standard deviation $\sigma = \sqrt{\sum_{i=1}^{n}(v_i - \mu)^2}$, then the following must be true: $\frac{\sigma}{\mu} \leq \varepsilon$. ε is column specific. ε is not defined in the specification. However, we can obtain ε for each column by calculating the coefficient of variation on the data generated by DBgen. PDGF data is then compliant if it yields a similar ε.

Row Cardinalities: The cardinalities of most tables depend on the scale factor or their cardinality is fixed. These are: Orders (SF*1,500,000), Customer (SF * 150,000), Supplier (SF * 10,000), Part (SF * 200,000) and Partsupp (SF * 800,000). The cardinalities of nation (25) and region (5) are scale factor independent. Verification of their cardinalities can be done with the SQL query listed in Figure 10, where T is the table name and S is its scaling relative to the scale factor SF.

```
SELECT CASE WHEN cnt=SF*S
            THEN 'OK' END
FROM (SELECT count(*) cnt
      FROM T);
```

Figure 10: Table cardinality compliance query

Table 2: Cardinalities DBgen and PDGF

Table	Table cardinalities @ SF=100		
	Specification	DBgen	PDGF
Orders	150 Million	150 Million	150 Million
Customer	15 Million	15 Million	15 Million
Supplier	1 Million	1 Million	1 Million
Part	20 Million	20 Million	20 Million
Partsupp	80 Million	80 Million	80 Million
Nation	25	25	25
Region	5	5	5

The cardinalities of Lineitem, on the other hand, depend on the cardinalities of other orders. To each row in the Orders table correspond a random number of rows within [1 .. 7] in the Lineitem table. More generally, to each row in the parent table P correspond n rows in the dependent table D. For dependencies like this three characteristics need to be analyzed i) Join Frequency. Given that each row of the parent table can join between 1 and 7 times to the dependent table, we need to calculate the range of join frequencies of parent to dependent rows ii) Coefficient of the frequency distribution, i.e. the distribution of how often rows of the parent table join to the dependent table is uniform. iii) Row counts. The following two SQL statements show how the relationship between Lineitem and Orders can be verified in SQL. Running these SQL statements on 100 SF databases, populated with DBgen and PDGF shows that the range of the join frequency is one to seven with roughly 21 Million records each and a coefficient of variation of 0.000197 for DBgen and 0.000002 of PDGF. The row count differs slightly. Dbgen generates 600,037,902 rows, while PDGF generates 600,000,000.

```
SELECT bucket
      ,bucketsize
      ,SUM(bucketsize) OVER
       (ORDER BY bucket ROWS
        BETWEEN UNBOUNDED PRECEDING
    AND CURRENT ROW) TotalBucketsize
FROM(SELECT bucket
           ,COUNT(*) bucketsize
     FROM (SELECT l_orderkey
                 ,COUNT(*) bucket
           FROM lineitem
               ,orders
           WHERE l_orderkey=o_orderkey
           GROUP BY l_orderkey)
     GROUP BY bucket);
```

```
SELECT stddev(bucketsize)
      /avg(bucketsize)
FROM(SELECT bucket
           ,COUNT(*) bucketsize
     FROM (SELECT l_orderkey
                 ,COUNT(*) bucket
           FROM lineitem
               ,orders
           WHERE l_orderkey
                =o_orderkey
           GROUP BY l_orderkey)
     GROUP BY bucket);
```

Date (min,max): The Date primitive generates a string of numeric characters separated by hyphens and comprised of a four digit year, two digit month and two digit day of the month, e.g. "1996-04-01". The TPC-H schema contains four date columns, L_Shipdate, O_Orderdate, L_Commitdate and L_Receiptdate. O_Orderdate is generated with a random date between Startdate and Enddate -151 days, while L_Shipdate, L_Commitdate and L_Receiptdate are generated by adding a random number as offset

```
SELECT MIN(O_Orderdate)
      ,MAX(O_Orderdate)
      ,count(distinct O_Orderdate)
FROM Orders;

SELECT STDDEV(c)/AVG(c)
FROM (SELECT O_Orderdate,count(*) c
      FROM Orders
      GROUP BY O_Orderdate);
```

Figure 11: Sample date column compliance query

between to O_Orderdate, i.e. l_shipdate = o_orderdate + random value [1 .. 121], l_commitdate = o_orderdate + random value [30 .. 90], l_receiptdate = o_orderdate + random value [1 .. 30].

To demonstrate compliance with the specification, we need to show three data characteristics for each date field i) the minimum and maximum dates are correct and iii) the date interval is dense, i.e. the number of distinct dates equals the number of dates between min and max and iii) the dates are uniformly distributed (see Figure 11).

Table 3: Comparison Date Distribution DBgen and PDGF

Column	CoV of dates		Date Range DBgen			Date Range PDGF		
	DBgen	PDGF	Min	Max	#distinct	Min	Max	#distinct
O_Orderdate	0.00388	0.00398	1992-01-01	1998-08-02	2406	1992-01-01	1998-08-02	2406
L_Shipdate	0.17970	0.17969	1992-01-02	1998-12-01	2526	1992-01-02	1998-12-01	2526
L_Commitdate	0.12762	0.12763	1992-01-31	1998-10-31	2466	1992-01-31	1998-10-31	2466
L_Receiptdate	0.20888	0.20887	1992-01-03	1998-12-31	2555	1992-01-03	1998-12-31	2555

Phone Number: The Phone Number primitive generates a string of numeric characters separated by hyphens and represented as follows: [1 .. 25]"-" [100 .. 999]"-" [100 .. 999]"-" [1000 .. 9999]. To demonstrate compliance with the specification, each of the four sections of the phone number needs to be investigated separately. For each we need to determine three characteristics i) the minimum and maximum values ii) the number of unique values and iii) whether the values are distributed uniformly. The phone number primitive applies to the fields S_Phone and C_Phone. The following four SQL statements, one for each section of the phone number field, show how the supplier phone number field S_Phone can be verified in SQL:

```
SELECT MIN(cc),MAX(cc),COUNT(*),STDDEV(cnt),STDDEV(cnt)/AVG(cnt)
FROM (SELECT SUBSTR(s_phone,1,2) cc,COUNT(*) CNT
      FROM supplier
      GROUP BY SUBSTR(s_phone,1,2));
SELECT MIN(cc),MAX(cc),COUNT(*),STDDEV(cnt),STDDEV(cnt)/AVG(cnt)
FROM (SELECT SUBSTR(s_phone,4,3) cc,COUNT(*) CNT
      FROM supplier
      GROUP BY SUBSTR(s_phone,4,3));
SELECT MIN(cc),MAX(cc),COUNT(*),STDDEV(cnt),STDDEV(cnt)/AVG(cnt)
FROM (SELECT SUBSTR(s_phone,8,3) cc,COUNT(*) CNT
      FROM supplier
      GROUP BY SUBSTR(s_phone,8,3));
SELECT MIN(cc),MAX(cc),COUNT(*),STDDEV(cnt),STDDEV(cnt)/AVG(cnt)
FROM (SELECT SUBSTR(s_phone,12,4) cc,COUNT(*) CNT
      FROM supplier
      GROUP BY SUBSTR(s_phone,12,4));
```

Figure 12: SQL statements to verify compliance of S_Phone data

Table 4: Comparison Phone fields DBgen and PDGF

	DBgen				PDGF			
Phone# Section	**MIN**	**MAX**	**Distinct**	**CoV**	**MIN**	**MAX**	**Distinct**	**CoV**
Country Code	10	34	25	0.0042	10	34	25	0.0039
Area Code	100	999	900	0.0293	100	999	900	0.0288
Phone # Part 1	100	999	900	0.0308	100	999	900	0.0301
Phone # Part 2	1000	9999	9000	0.0952	1000	9999	9000	0.0950
Country Code	10	34	25	0.0011	10	34	25	0.0013
Area Code	100	999	900	0.0076	100	999	900	0.0078
Phone # Part 1	100	999	900	0.0077	100	999	900	0.0082
Phone # Part 2	1000	9999	9000	0.0245	1000	9999	9000	0.0246

Table 5: CoV of Random String values

Column	**CoV**		**Idential List_name**
	DBgen	**PDGF**	
P_Type	0.00280	0.00293	Yes
P_Container	0.00142	0.00131	Yes
C_Mktsegment	0.00062	0.00054	Yes
L_Shipinstruct	0.00008	0.00012	Yes
L_Shipmode	0.00011	0.00012	Yes
O_Orderpriority	0.00012	0.00009	Yes

Random String (list_name): The Random String primitive generates a string selected at random within a list of strings (list_name). Each string is selected with equal probability. It applies to columns P_Type, P_Container, C_Mktsegment, L_Shipinstruct, L_Shipmode and O_Orderpriority. For each of these fields we need to verify the following two data characteristics i) The distinct elements in the column correspond to the TPC-H specific and ii) the distribution of the elements is random. Figure 13 shows an example how to verify i) and ii) for C_Mktsegment using SQL and Table 5 shows the results for running this type of SQL for all Random String columns. For a list of SQL statements for all other columns using the Random String primitive, please refer to Appendix B

```
SELECT UNIQUE C_Mktsegment
FROM CUSTOMER;

SELECT STDDEV(cnt)/AVG(cnt)
FROM (SELECT C_Mktsegment mseg
            ,COUNT(*) cnt
      FROM customer
      GROUP BY C_Mktsegment);
```

Figure 13: SQL to verify compliance of C_Mktsegment data

Random Value (min,max): The Random Value primitive defines random values between min and max inclusively, with a mean of (min+max)/2. The columns using this primitive are P_Size, Ps_Availqty, Ps_Supplycost, C_Acctbal, L_Partkey, C_Nationkey L_Discount, L_Tax, L_Quantity, S_Nationkey and S_Acctbal. For each column we need to verify three characteristic i) min and max values ii) number of distinct values and iii) coefficient of variation of the value probabilities. The following two SQL statements verify S_Nationkey column, a foreign key to the Nation table in Supplier. For a list of SQL statements for all columns using the Random Value primitive, please refer to Appendix B.

```
SELECT MIN(S_Nationkey),MAX(S_Nationkey),COUNT(DISTINCT S_Nationkey)
FROM Supplier;

SELECT STDDEV(cnt)/AVG(cnt)
FROM (SELECT S_Nationkey,COUNT(*) cnt
      FROM SUPPLIER GROUP BY S_Nationkey);
```

Figure 14: SQL statements to verify compliance of S_Nationkey data

Table 6: CoV or Random Values

	DBgen				PDGF			
Column	**MIN**	**MAX**	**Distinct**	**CoV**	**MIN**	**MAX**	**Distinct**	**CoV**
P_Size	1	50	50	0.00151	1	50	50	0.00149
Ps_Availqty	1	9999	9999	0.01083	1	9999	9999	0.10003
Ps_Supplycost	1.00	1000.00	99901	0.03469	1.00	1000.00	99897	0.31573
C_Acctbal	-999.99	9999.99	1099998	0.26985	-999.99	9999.99	1099999	0.27089
L_Partkey	1	20e6	20e6	0.15503	1	20e6	20e6	0.18264
C_Nationkey	0	24	25	0.00105	0	24	25	0.00110
L_Discount	0	0.1	11	0.00011	0	0.1	11	0.00012
L_Tax	0	0.08	9	0.00007	0	0.08	9	0.00014
L_Quantity	1	50	50	0.00028	1	50	50	0.00032
S_Nationkey	0	24	25	0.0042	0	24	25	0.00519
S_Acctbal	-999.99	9999.98	656803	0.50379	-999.99	9999.98	656803	0.50393

Random v-String: A Random v-String primitive represents a string comprised of randomly generated alphanumeric characters within a character set of at least 64 symbols. The length of the string is a random value between values min and max inclusive. Columns using this data generation primitive are the address columns: C_Address, S_Address, C_Address. For each column we need to determine three data characteristics i) domain over which the strings are generated ii) are the strings picked randomly? iii) min, max length of each string and distribution of length across all fields. i) can be determined with SQL in Figure 15. ii) can be determined with the SQL in

```
SELECT SUM(LENGTH(S_Address)-LENGTH(REPLACE(S_Address,CHR(1),''))) S0
    ,SUM(LENGTH(S_Address)-LENGTH(REPLACE(S_Address,CHR(1),''))) S1
    ,…
    ,SUM(LENGTH(S_Address)-LENGTH(REPLACE(S_Address,CHR(1),''))) S255
FROM SUPPLIER;
```

Figure 15: SQL statement to verify compliance of v-String data

```
SELECT STDDEV(Col)/AVG(Stddev)FROM(
SELECT SUM(LENGTH(s_address)-LENGTH(REPLACE(s_address,CHR(1),''))) COL
FROM SUPPLIER
UNION ALL
      ,SUM(LENGTH(s_address)-LENGTH(REPLACE(s_address,CHR(1),''))) COL
FROM SUPPLIER
UNION ALL
,…
SELECT SUM(LENGTH(s_address)-LENGTH(REPLACE(s_address,CHR(1),''))) COL
FROM SUPPLIER);
```

Figure 16: SQL compliance query to determine random distribution of characters

```
Select min(length(s_address)
      ,max(length(s_address)
      ,stddev(length(s_address)/avg(length(s_address)
From supplier;
```

Figure 17: SQL compliance query to determine minimal, maximal length and length distribution

Text Appended with Digit: The Text Appended with Digit primitive represents a string generated by concatenating a sub-string text with a number. Columns using this primitive are S_Name, C_Name, P_Mfgr, P_Brand and O_Clerk. Columns S_Name and C_Name append the content of another column (of the same row), while p_Mfgr,

P_Brand and O_Clerk append a random number within *min* and *max*. To demonstrate compliance with the specification of columns appending the value of another column, we need to demonstrate that the appended value is equal to the value of the other column. Figure 18 shows a compliance query that counts the number of rows where the values are different. A result of 0 indicates compliance with the specification. To demonstrate compliance with the specification of columns adding a random number, we need to demonstrate three data characteristics i) the minimum and maximum values of the appended number correspond to the *min* and *max* values of the specification; ii) the number of distinct values and iii) the values are distributed randomly. Figure 19 shows a compliance query that computes the minimum and maximum values, the number of distinct values and the coefficient of variation of the distribution of the values between minimum and maximum. For a list of all columns using the Text Appended with Digit primitive, please refer to Appendix **Error! Reference source not found.**.

Table 7: CoV of Text Appended with Digit values

Column	CoV		Specification		PDGF	
	DBgen	PDGF	min/max		min/max	
O_Clerk	0.02587	0.02587	1	100000	1	100000
P_Mfgr	0.00031	0.00033	1	5	1	5
P_Brand	0.00044	0.00043	1	5	1	5

```
select sum(case when s_suppkey = col then 0 else 1 end)
from (select to_number(substr(s_NAME,10,length(s_NAME)-9)) col
            ,s_suppkey
      from supplier);
```

Figure 18: Compliance query for Text with Append Digit primitive (column)

```
select min(col)
      ,max(col)
      ,count(*)
      stddev(cnt)/avg(cnt)
from (select to_number(substr(o_clerk,10,length(o_clerk)-9)) col
            ,count(*) cnt
    from orders
    group by to_number(substr(o_clerk,10,length(o_clerk)-9)));
```

Figure 19: Compliance query for Text with Append Digit primitive (random number)

Unique Value (min,max): The Unique Value primitive generates unique values between 1 and x. Columns using this primitive are S_Suppkey [1 .. sf * 10,000], P_Partkey [1 .. sf * 200,000], C_Custkey [1 .. sf * 150,000], N_Nationkey [0 .. 24], R_Regionkey [0 .. 4] and O_Orderkey [1 .. sf * 1,500,000 * 4]. O_Orderkey has an additional requirement, as only the first 8 keys of each 32 are to be populated. For each column we have to verify four data characteristics i) minimum value ii) maximum values iii) number of distinct values and iv) number of rows. Data is generated correctly if the minimum and maximum values correspond to the specification and the number of distinct values equals the number of rows. The following table lists the result of the Query listed in Figure 20 for Dbgen and PDGF.

```
SELECT MIN(S_Suppkey)
      ,MAX(S_Suppkey)
      ,COUNT(DISTINCT S_Suppkey)
      ,COUNT(S_Suppkey)
FROM Supplier;
```

Figure 20: SQL to verify Unique Values

Table 8: Results of Unique Value tests for all affected columns

	TPC-H Specification (@SF=100)			PDGF			
Column	**MIN**	**MAX**	**Distinct**	**MIN**	**MAX**	**Distinct**	**Count**
S_Suppkey	1	1000000	1000000	1	1000000	1000000	1000000
P_Partkey	1	20000000	20000000	1	20000000	20000000	20000000
C_Custkey	1	15000000	15000000	1	15000000	15000000	15000000
N_Nationkey	0	24	25	0	24	25	25
R_Regionkey	0	4	5	0	4	5	5
O_Orderkey	1	6E+08	1.5E+08	1	6E+08	1.5E+08	1.5E+08

```
SELECT COUNT(*)
FROM (SELECT MOD(O_ORDERKEY,9)
MODVALS
      FROM ORDERS )
WHERE MODVALS <= 8;
```

Figure 21: SQL to check the sparsely populated O_Orderkey

In addition to the above, for O_Orderkey we need check that the keys are only sparsely populated, i.e. only the first 8 keys of every 32 keys are used. The following SQL statement counts the number of keys that fall in residue classes small or equal than 8. If this query returns the same number of rows than the total number of rows (last column in Table 5), then only the first 8 keys of every 32 keys are populated. For a list of all columns using the Unique Value primitive see Appendix B.

Random Text Strings (min,max) is a pseudo English text generated over a fixed dictionary following the grammar defined in Clause 4.2.2.14. In order to assure that the text was generated with the grammar in Clause 4.2.2.14 one would need to write a parser for the grammar. Since the grammar of the text is not exploited in the benchmark, the authors believe that by checking i) the minimum length ii) the maximum length and iii) the uniqueness and the uniform distribution of the length, suffices to assure compliance with the specification. SQL in Figure 22 checks this.

Table 9: CoV of Random Text Strings

Column	CoV		Spec		PDGF	
	DBgen	**PDGF**	**min/max**		**min/max**	
L_Comment	0.00024	0.00023	10	43	10	43
O_Comment	0.00057	0.00032	19	78	19	78
S_Comment	0.00850	0.00477	25	100	25	100
P_Comment	0.00087	0.00056	5	22	5	22
PS_Comment	0.00124	0.00073	49	198	49	198
C_Comment	0.00247	0.00143	29	116	29	116
N_Comment	0	0	31	114	31	114
R_Comment	0.4	0.4	31	115	31	115

```
SELECT MIN(l)
      ,MAX(l)
      ,STDDEV(c)
       /AVG(c)
FROM(SELECT
     LENGTH(L_Comment)l
    ,COUNT(*) c
     FROM Lineitem;
```

Figure 22: SQL to check compliance of L_Comment

4.1 Special Cases

O_Shippriority should be set to 0 for all orders. Compliance with the TPC-H specification can be checked with the following simple SQL query:

```
SELECT CASE WHEN c=0 THEN 'OK' END FROM (SELECT count(*) c from orders);
```

The above query shows OK for DBGen and PDGF on a SF=100 database.

L_Returnflag is set to "R" or "A" if L_Receiptdate <= Currentdate. Otherwise it is set to "N". The following SQL query counts the number of rows with L_Returnflag

equal to R, A and N when L_Receiptdate is less or equal than currentdate and when L_Receiptdate is greater than currentdate. If the following SQL query returns 0 for lessAndN, largerAndR and largerAndA, then L_Returnflag conforms to TPC-H:

```
SELECT SUM(CASE WHEN L_Receiptdate<=TO_DATE('1995-06-17','YYYY-MM-DD')
                 AND L_Returnflag = 'R' THEN 1 ELSE 0 END) lessAndR
       ,SUM(CASE WHEN L_Receiptdate<=TO_DATE('1995-06-17','YYYY-MM-DD')
                 AND L_Returnflag = 'A' THEN 1 ELSE 0 END) lessAndA
       ,SUM(CASE WHEN L_Receiptdate<=TO_DATE('1995-06-17','YYYY-MM-DD')
                 AND L_Returnflag = 'N' THEN 1 ELSE 0 END) lessAndN
       ,SUM(CASE WHEN L_Receiptdate>TO_DATE('1995-06-17','YYYY-MM-DD')
                 AND L_Returnflag = 'R' THEN 1 ELSE 0 END) largerAndR
       ,SUM(CASE WHEN L_Receiptdate>TO_DATE('1995-06-17','YYYY-MM-DD')
                 AND L_Returnflag = 'A' THEN 1 ELSE 0 END) largerAndA
       ,SUM(CASE WHEN L_Receiptdate>TO_DATE('1995-06-17','YYYY-MM-DD')
                 AND L_Returnflag = 'N' THEN 1 ELSE 0 END) largerAndN
       ,COUNT(*) CNT
from lineitem;
```

Figure 23: SQL compliance query for L_Returnflag

The above query shows returns 0 for lessAndN, largerAndR and largerAndA for DBGen and PDGF on a SF=100 database. DBGen and PDGF show a count of 0 for this query.

P_Retailprice is set to P_Retailprice = (90000 + ((P_Partkey/10) modulo 20001) + 100 * (P_Partkey modulo 1000))/100. The following SQL query counts the number of rows where P_Retailprice is not computed correctly:

```
SELECT SUM(CASE WHEN P_Retailprice-(90000+(MOD((P_Partkey/10),20001))
                                    +100*(MOD(P_Partkey,1000)))/100
                THEN 1 ELSE 0 END) cnt
from part;
```

Figure 24: SQL compliance query of P_Retailprice

L_Linestatus is set to "o" if l_Shipdate > currentdate, to "f" otherwise. The following SQL query counts the correct cases where L_Linestatus should be set to O and F. If the sum of largerAndO and lessOrEqualAndF equals cnt, then L_Linestatus conforms to the TPC-H specification. DBGen and PDGF show pass this query test on a SF=100 database.

```
SELECT SUM(CASE WHEN L_Shipdate >TO_DATE('1995-06-17','YYYY-MM-DD')
                 AND L_Linestatus='O'THEN 1 ELSE 0 END)largerAndO
       ,SUM(CASE WHEN L_Shipdate <=TO_DATE('1995-06-17','YYYY-MM-DD')
                 AND L_Linestatus='F'THEN 1 ELSE 0 END)lessOrEqualAndF
       ,COUNT(*) cnt
From Lineitem;
```

Figure 25: SQL compliance query for L_Linestatus

P_Name is generated by concatenating five unique randomly selected strings from a list of colors (see Clause 4.2.3 of the TPC-H specification for details). Verifying P_Name is not straight forward in SQL. One needs to extract all five colors and then check pair wise for duplicates. The following SQL query counts the number of rows with duplicate colors in P_Name. A result of 0 signifies compliance with the TPC-H specification. Both Dbgen and PDGF show 0 for this query.

```
SELECT SUM(CASE WHEN C1=C2 OR C2=C3 OR C3=C4 OR C4=C5 OR C2=C3
                     OR C2=C4 OR C2=C5 OR C3=C4 OR C3=C5 OR C4=C5
                THEN 1 ELSE 0 END)
FROM (SELECT SUBSTR(P_Name,1,SA-1) C1,SUBSTR(P_Name,SA+1,SB-SA) C2
            ,SUBSTR(P_Name,SB+1,SC-SB) C3,SUBSTR(P_Name,SC+1,SD-SC) C4
            ,SUBSTR(P_Name,SD+1,LENGTH(P_Name)-SD+1) C5
      FROM (SELECT P_Name
                  ,INSTR(P_Name,' ',1,1) SA ,INSTR(P_Name,' ',1,2) SB
                  ,INSTR(P_Name,' ',1,3) SC ,INSTR(P_Name,' ',1,4) SD
            FROM Part));
```

Figure 26: SQL compliance query for P_Name

O_Totalprice is computed as sum(L_Extendedprice*(1+L_Tax)*(1-L_Discount)) for all Lineitem of this order. In order to verify O_Totalprice we need to join Orders with Lineitem and calculate the sum. The following SQL query verifies this for all rows:

```
SELECT COUNT(*)
FROM(SELECT O1.O_Orderkey OK, SUM(L1.L_Extendedprice
                              *(1+L1.L_Tax)*(1-L1.L_Discount)) TP
     FROM Lineitem L1,Orders O1
     WHERE L1.L_Orderkey=O1.O_Orderkey
     GROUP BY O1.O_Orderkey),Orders O2
WHERE OK<>O2.O_Orderkey And O2.O_Totalprice<>TP;
```

N_Nationkey, N_Name, N_Regionkey is statically to a list of combinations. This list of combinations is defined in Clause 4.2.3. Both DBgen and PDGF generate a correct set of combinations.

R_Regionkey, R_Name is statically to a list of combinations. This list of combinations is defined in Clause 4.2.3. Both DBgen and PDGF generate a correct set of combinations.

Ps_Suppkey defined as (PS_Partkey+(i*((S/4)+(int)(PS_Partkey-1)/S)))) modulo S+1, where i is the i-th supplier within [0 .. 3] and S = SF * 10,000. The following verifies compliance of PS_Suppkey for scale factor 100. If the values of Matching and Cnt are identical PS_Suppkey is generated in compliance with the specification. Both DBgen and PDGF generate compliance data for PS_Suppkey.

```
SELECT SUM (CASE WHEN (Ps_Suppkey=MOD(Ps_Partkey+0*((1000000/4)
                           +(TRUNC((Ps_Partkey-1)/1000000))),1000000)+1)
                  OR (Ps_Suppkey=MOD(Ps_Partkey+1*((1000000/4)
                           +(TRUNC((Ps_Partkey-1)/1000000))),1000000)+1)
                  OR (Ps_Suppkey=MOD(Ps_Partkey+2*((1000000/4)
                           +(TRUNC((Ps_Partkey-1)/1000000))),1000000)+1)
                  OR (Ps_Suppkey=MOD(Ps_Partkey+3*((1000000/4)
                           +(TRUNC((Ps_Partkey-1)/1000000))),1000000)+1)
                THEN 1 ELSE 0 END) Matching
       ,COUNT(*) Cnt FROM Partsupp;
```

Figure 27: SQL compliance query for Ps_Suppkey

5 Conclusion

In this paper, we have shown that TPC-H equivalent data can be generated with the generic data generator PDGF. First we analyzed the generation requirements of TPC-

H data and showed how they can be implemented using PDGF. The complete configuration file for PDGF is given as supported material to this paper. To proof that our PDGF implementation is compliant with the current TPC-H specification (Version 2.14.2), we first developed a mathematical way to determine compliance based on the coefficient of variation, minimum, maximum values, among others. We also provided SQL statements to calculate these values. Examples of these statements are given in the paper, while a complete list is given as supporting material. Using scale factor 100, we generated a complete data set with both DBgen and PDGF.
Running the compliance queries on the scale factor 100 database showed that both tools generate data that is compliant with the specification. All minimum, maximum values and distributions in general are identical between the two tools. One of the major characteristics of TPC-H's data is that it is distributed uniformly. This is very important as the benchmark's execution rules rely on it. Our indicator for uniform distribution has been defined as the coefficient of variation (CoV). DBgen shows a wide range for the CoV of various colums. For instance, the CoV of the distribution of lineitem to orders is 0.000197 while the CoV of L_Partkey is 0.15503. It is up to the TPC to decide whether these CoV are specification conforming. For the sake of this paper, however, it is only important whether the data PDGF generates has the same or better CoV. Our data shows that in most cases the CoV of PDGF data is better than that of DBgen data. Only in a few cases, DBgen generates data with a lower CoV. For instance, Ps_Supplycost shows a CoV of 0.31573 with PDGF and 0.03469 with DBGen. In time for the completion of this paper we were not able to fully investigate these cases. We hope to have completed this work by the time of the workshop.

Apart from data generation itself, PDGF has many advantages over DBgen. Since it is written in the platform independent language Java, it is very portable to new, emerging platforms. Due to its popularity, Java programming expertise is high amongst contractors and companies alike. Its generic nature, i.e. its separation into a data generation engine and a file defining the metadata about the data to be generated also suggests that PDGF could be the default data generator for the TPC. This will reduce development cost of new benchmarks and maintenance cost of existing benchmarks. Finally, previous studies have shown that PDGF is able to generate terabytes of data quicker than tools currently deployed by the TPC [2].

6 Acknowledgements

The authors would like to acknowledge Ray Glasstone for his support. Thanks to John Galloway for reviewing early versions of the TPCH.pdgf implementation.

7 References

1. M. Poess and C. Floyd, 2000, *New TPC Benchmarks for Decision Support and Web Commerce*. ACM SIGMOD RECORD, 29(4), pp. 64-71
2. T. Rabl, M. Frank, H. Mousselly Sergieh, H. Kosch. 2010. *A Data Generator for Cloud-Scale Benchmarking.* TPCTC '10, 41-56. LNCS 6417. Springer.
3. T. Rabl and M. Poess. 2011. *Parallel Data Generation for Performance Analysis of Large, Complex RDBMS.* DBTest '11
4. J. M. Stephens, M. Poess, 2004, *MUDD: a multi-dimensional data generator*. WOSP '04, pp. 104-109
5. TPC-D, Version 2.1, http://www.tpc.org/tpcd/default.asp
6. TPC-H Specification, Version 2.14.0, http://www.tpc.org/tpch/spec/tpch2.14.0.pdf