Parallel Data Generation for Performance Analysis of Large, Complex RDBMS

Tilmann Rabl
Universität Passau
Innstraße 43
94032 Passau
49-851-509-3067
rabl@fim.uni-passau.de

Meikel Poess
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA-94065
1-650-633-8012
meikel.poess@oracle.com

ABSTRACT
The exponential growth in the amount of data retained by today's systems is fostered by a recent paradigm shift towards cloud computing and the vast deployment of data-hungry applications, such as social media sites. At the same time systems are capturing more sophisticated data. Running realistic benchmarks to test the performance and robustness of these applications is becoming increasingly difficult, because of the amount of data that needs to be generated, the number of systems that need to generate the data and the complex structure of the data. These three reasons are intrinsically connected. Whenever large amounts of data are needed, its generation process needs to be highly parallel, in many cases across-systems. Since the structure of the data is becoming more and more complex, its parallel generation is extremely challenging. Over the years there have been many papers about data generators, but there has not been a comprehensive overview of the requirements of today's data generators covering the most complex problems to be solved. In this paper we present such an overview by analyzing the requirements of today's data generators and either explaining how the problems have been solved in existing data generators, or showing why the problems have not been solved yet.

Categories and Subject Descriptors
H.3.4 [**Information Storage and Retrieval**]: Systems and Software - *Performance evaluation (efficiency and effectiveness)*

General Terms
Algorithms, Measurement, Performance, Experimentation.

Keywords
Parallel data generation, pseudorandom number generation, database benchmarking.

1. INTRODUCTION
The exponential growth in the amount of data retained by today's systems is fostered by a recent paradigm shift towards cloud computing and the vast deployment of data-hungry applications, such as social media sites. The amount of data Facebook collected grew exponentially from 15TByte in 2007 to 700TByte in 2010 [14]. At the same time systems are capturing more sophisticated data. In these cases data may show complex dependencies. Normalized schemas try to avoid data dependencies. However, even highly normalized schemas have dependencies. In order to increase performance for analytical processing, less normalized or highly de-normalized schemas are used, such as in data warehouses or online analytical processing (OLAP) systems. These systems show many different types of dependencies.

Running realistic benchmarks to test the performance of these systems is becoming increasingly difficult, because of the amount of data that needs to be generated, the number of systems that need to participate in the data generation and the complex structure of the data. Especially for system robustness testing it is necessary to drive systems to the edge of their limits in terms of the amount of data and the complexity of the data. Hence, the requirements to modern data generators have increased over time. They are required to generate data

- Deterministically,
- In parallel processes/threads,
- In parallel across address space of multiple systems and
- Generate complex data set.

Yet data generators need to adapt to different schemas and need to be easy to set up. Generating data deterministically is one of the most important requirements, since without it, comparing results from multiple benchmark runs is very difficult. If data is not generated deterministically, one would need to proof that different data sets result in the same workload. This is difficult to proof on different hardware/software platforms, because the impact of different data sets on performance may depend on the hardware and software architecture. Hence, data generators must generate the exact same data set regardless of the degree of parallelism and hardware architecture.

Data generation speed is also important since today's systems need to test for very large amounts of data. This requires the programmer to take advantage of multi-core processors and large clusters of systems. Hence, data needs to be generated in parallel threads across address space boundaries.

If the hurdle of generating and using data is too high, the acceptance of a benchmark is low. That is, the generation process should be simple and adaptive to allow for easy data generation and different forms of output.

Benchmarks specifications try to model their domain as realistically as possible. "Time to market" is a crucial aspect in bench-

DBTest '11, June 13, 2011 Athens, Greece

mark design, because they are vulnerable to becoming outdated very quickly [13]. An important feature of a good data generator is, therefore, that it can be adapted to changing requirements in a simple way.

In order to test the fine nuances of systems for a specific application domain, one needs to generate data with certain properties. For instance, in order to measure the performance of Extract Transform and Load (ETL) system's ability to maintain history changing dimensions, the data to feed the ETL system needs to follow certain patterns, i.e. address changes occur in a certain time order, account balances are computed based on transaction patterns. These data characteristics can be formalized as *intra row*, *intra table* and *inter table* dependencies. Without the requirements of deterministic data generation, parallel data generation (across cores/ across address space of multiple systems), simplicity and adaptability, the generation of complex data has been solved in previously published work on data generators, we will present different approaches in the related work in Section 2. However, in the presence of these requirements efficient data generation is extremely difficult and has not been sufficiently studied.

In this paper we present an overview of the requirements on generating complex data and how it can be generated efficiently. After we give an overview of related work in Section 2 we outline the fundamentals of modern data generators, namely the use of pseudo random numbers in parallel environments in Section 3. In Section 4 we develop requirements of modern data generators by analyzing the data used in some of today's well known standard benchmarks and important future benchmarks in areas such as ETL. In Section 5 we present how most of the identified requirements can be implemented in the parallel data generation framework PDGF presented in [11] and continue with an analysis of those requirements that cannot be implemented yet in PDGF before we conclude our paper in Section 6.

2. RELATED WORK

There has been a considerable amount of research published on data generation for benchmarking purposes. Gray et al. have shown how to generate large data sets with statistical distributions and dense unique sequences in parallel [4]. This work has been the basis for other works on parallel generation of uncorrelated data.

Data generation for performance evaluation is part of the daily business of researchers and DB administrators. Since most of their data generators are special purpose implementations for a single dataset/ single test, developing these them is very time consuming and costly. Alternatively, some use data generators that were developed by standard benchmark organizations, such as TPC and SPEC or in open projects such as YCSB [3]. However, these data generators are also special purpose implementations, which can be adapted only to a limited degree. The existing generic data generators can be divided into two categories: a) Simulation of client interaction, and b) Usage of synthetic statistical distributions.

Usually generic data generators that simulate client data are based on graphs. Houkjær et al. describe such a graph based generation tool [6]. It uses a depth-first graph traversal to generate dependent tables; cyclic dependencies are resolved by creating temporary values. The generated data is directly written to a database and read again for reference generation. A similar graph based approach was presented by Lin et al. [7]. The largest advantage of such generators is their possibility of generating data according to very complex and realistic workflows. The downside is a very slow generation speed that limits the usage to small data sets.

Data generators based on synthetic data are based on statistical distributions. Examples are MUDD by Stephens and Poess [12] and PSDG by Hoag et al. [5]. Both generators feature a description language to describe the database schema and the data characteristics. Furthermore, both data generators are able to parallelize data generation. MUDD uses a platform independent 64 bit linear congruential random number generator that, utilizing partition-wise data generation and dense-unique-pseudo-random sequences, enables it to generate in parallel deterministic, randomized real world data. In order to generate consistent references, PSDG queries the generated data. The data has to be written directly to a single database system, which is then queried by the generator instances. Hence, the data generation speed is limited by the insert and access rate of the database system. A related approach was presented by Bruno and Chaudhuri [1]; it largely relies on scanning a given database to generate various distributions and interdependencies. Although the bootstrapping process from a given database is user friendly, this form of generation is very slow and is parallelized only partly.

3. PARALLEL RNGs

The basis of synthetic data generation is the use of random number generators (RNG). In most cases deterministic data generation is necessary to allow repeatable experiments on various kinds of systems. Therefore, pseudo random number generators are used, which allow deterministic generation of random looking data. Many different kinds of pseudo random number generators have been proposed. They differ highly in the statistical quality of their data and the length of their period. The quality of random numbers is important to avoid the creation of patterns. Quality can be measured with the Die Hard Battery of Tests of Randomness [9]. Equally important for the generation of very large data sets is the period length of the pseudo random number stream. Every generator eventually repeats its random number stream. It therefore has to be assured, that the period is large enough for the amount of data to be generated.

In order to generate data for large data sets, it is desired to generate data in parallel. For this reason parallel pseudo random number generators have been developed. In contrast to linear generators, parallel generators do not reseed the data generator after every number. A linear generator calculates a random number as follows: $rng(n) = lrng(lrng(\dots(lrng(seed))\dots))$. Efficient implementation of linear pseudo random number generators store the last generated value as their internal state. Parallel random number generators, however, are often stateless by generating random numbers independently using a hash function: $rng(n) = prng(n + seed)$. Examples can be found in [10]. Using this approach, the generation of random number sequences can easily be distributed to many parallel processes. In order to generate the same sequence on different numbers of processors, either the leapfrog method or sequence splitting is used [2]. While the leapfrog method partitions the sequence in turn between processes, sequence splitting partitions the sequence in contiguous sequences. The concrete value generation is a function from the random number to a value. An example of a generator for realistic data is a name generator based on a dictionary lookup. To gener-

ate a name two entries of a list of common first and last names[1] can picked based on a random number.

Usually random number generators generate uniformly distributed values. However more natural distributions can be calculated easily. For normal distributions the Box-Muller method and the related, usually faster polar method, can be used [8].

4. DATA REQUIREMENTS

In this section we motivate various data characteristics that are imperative to generating the data sets for large, complex database management systems. We introduce each topic by listing a couple of examples before formalizing the data characteristic. For the formalization we use the following simplified notation. The data generator generates one file F_R per database table $R\{C_1, C_2, \ldots, C_n\}$, where C_i denotes column i. Each file contains n rows, delimited by some character D. Row j is denoted as R_j with $j>0$. Rows are sorted, i.e. $R_j < R_{j+k}$ for $j,k>0$. A specific value of R is denoted as $R_n\{C_m\}$, where n is the row in file F_R and m is the column.

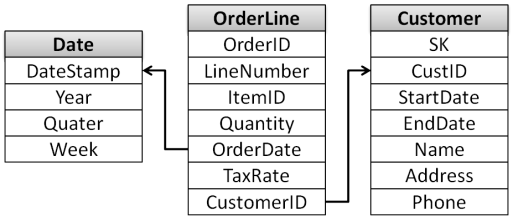

Figure 1: Sample data warehouse schema

Our standard example is shown in Figure 1; it depicts a simple data warehouse scenario. The relation *OrderLine* stores all order positions. The relation *Customer* stores all information about customers and *Date* is a dimensional table for dates. The following list of data characteristics is focused on common data dependencies, grouped in *intra row*, *intra table* and *inter table* dependencies. It does not claim to be exhaustive.

4.1 Intra Row Dependency

Intrinsic to de-normalized data warehouse schemas is intra row dependency. That is, some fields of the same row may exhibit some sort of dependencies. For instance, in the US, value added tax (VAT) varies by state and within some states by county. Hence, the VAT depends on the location of the purchase. Intra row dependencies are also very common in hierarchies, which are key features in navigating dimensions of star and snowflake schemas. For instance, a date dimension might have multiple hierarchies, such as the calendar hierarchy, which starts at the day and rolls up into month, quarter and year. There is an obvious dependency between these fields. Often there are other hierarchies due to companies' fiscal calendar not being aligned with the year calendar. Sample data using the date relation introduced in Figure 1 can be found in the Figure 2 below.

Date			
DateStamp	**Year**	**Quarter**	**Week**
2011-03-30	2011	201101	2011W13
2011-03-31	2011	201101	2011W13
2011-04-01	2011	201102	2011W13

Figure 2: Intra row dependencies in the date hierarchy

[1] See http://www.census.gov/genealogy/www/ for US names

Year, *Quarter* and *Week* are dependent on the entry in the column *DateStamp* This intra row dependency can be formally expressed as:$\{DateStamp\} \rightarrow \{Year, Quater, Week\}$

In database theory intra row dependencies are called functional dependencies. They can exist between two subsets of attributes of a relation. Obviously, there always exists a functional dependency from any key of a relation to all attributes. In general intra row dependency of a set of columns X to a set of columns Y of relation R can be expressed as: $R_n\{X\} \rightarrow \{Y\} : n > 0$

4.2 Intra Table Dependency

In addition to intra row dependencies data warehouses often exhibit intra table dependencies. That is, values of different rows within the same table have dependencies.

OrderLine				
OrderID	**LineNumber**	**ItemID**	**Quantity**	**...**
123	1	43	3	...
123	2	66	50	...
123	3	75	1	...

Figure 3: Typical denormalized Order/Lineitem relationship

One type of such dependencies can be found in the n:1 relationships of normalized schemas and their de-normalized counterparts. For instance, consider the traditional *order* and *lineitem* scenario or a retailer, frequently used in benchmarks, e.g. TPC-C and TPC-H [13]. Usually customers buy between one and n items per order. Hence, in a normalized schema orders are stored in an *order* table and its corresponding items are stored in the *lineitem* table. A de-normalized version merges the two tables into one *OrderLine* table (see Figure 3). Usually benchmark configurations mandate a specific distribution of lineitems per order with *min* being the minimum and *max* being the maximum number of lineitems per order. Hence, the *OrderLine* table contains dependencies between multiple rows, in this case rows grouped by *OrderID*. That is, there are between *min* and *max* number of rows with the same *OrderID*, but different *LineNumbers*.

A more complex example is that of history-keeping dimensions. History-keeping dimensions keep information about data changes or entities, such as customers or parts. In order to distinguish between current and historic data, both the "natural" (a.k.a. "business") key for an entity and its surrogate key are stored in the same row. The business key is the primary key from the system the row was extracted from. Examples are customer id and purchase number. Surrogate keys, being generated, are independent of the business function. The fact tables that need to reference a history-tracking dimension include a foreign key reference to the surrogate key, not the natural key. Additionally, each dimension entry has usually a start and end date, which indicate the time period for which records are valid. The most current one usually contains a special value (e.g. NULL) as the end date. Therefore data that reflect multiple incremental loads of a history keeping dimensions needs to have:

1. Monotonically increasing numbers in surrogate key columns,
2. Monotonically increasing date values in the start and end date,
3. End date after begin date, except for the current row and
4. The begin date of a row be equal to the end date of its predecessor, except for the current row.

Customer					
SK	CustID	StartDate	EndDate	Name	…
1	234	03-01-09	01-02-10	Smith	…
2	123	04-04-04	NULL	Wilson	…
3	234	01-02-10	NULL	Smith	…

Figure 4: Sample history keeping customer table

Figure 4 shows a history keeping customer table. It contains entries for two people Smith and Wilson. This example exhibits multiple dependencies that stem from the timeline the data was entered into the dimensions. This can be solved by sorting data during their generation. Obviously, surrogate keys are generated as increasing sequence. Furthermore, the start and end date are increasing for a single customer ID. And finally, the end date in a certain row is the start date of the next row with the identical business key. We can distinguish between regular and irregular sort orders. A regular sort order has the form of a sequence. So each value is dependent on the position. While an irregular order just requires that successive values follow the sort order. Example of data that is generated in regular order is the generation of surrogate keys, which are usually implemented as a sequence of integers or timestamps of regular events. An example of data that is generated following an irregular order is the generation of timestamps of random events such as address changes. However, keep in mind that benchmarks would like to generate them according to a specific distribution. The order can be in form of a dependency. This is shown in the example above. The start date of an entry in the history keeping dimension is the end date of the preceding entry with same business key.

Customer					
SK	CustID	…	Name	Address	Telephone
1	1	…	Smith	543 Arch	555-23
2	1	…	Smith	543 Arch	555-44
3	1	…	Smith	67 Second	555-23
4	1	…	Smith	67 Second	555-44
5	2	…	Wilson	2 Second	555-67

Figure 5: Multi valued dependency

Yet another form of intra table dependency stems from schema de-normalization. If a customer has several telephone numbers, multiple rows per customer exist with some data being duplicated in the customer table. Furthermore, if a customer also has several addresses all combinations of telephone numbers and addresses have to be stored. This is also called multi-valued functional dependency: $C\{id\} \rightarrow\rightarrow C\{address\}$, $C\{id\} \rightarrow\rightarrow C\{telephone\}$. The reason for multi-valued functional dependencies can also be poor schema design. In this case benchmarks concerned with the testing of database robustness in the context of odd or poor schema design want to inject such dependencies into their data.

In some cases aggregate data is kept for performance reasons. For instance, a fact table may record customer account transactions. For performance reasons the fact table entry recording the transactions might record the customer account balance after the money was withdrawn/deposited. This type of dependency is between the current account balance, the current transaction amount and the previous account balance. Other aggregation types such as the average account balance across all customers have dependencies to all preceding transactions of a customer.

4.3 Inter Table Dependency

The classical example of inter table dependency is referential integrity (RI). If two tables have a parent/child relationship, then for each value in the foreign key column of the parent table a primary key value in the child table must exist. For instance, for each customer in the *OrderLine* table (see Figure 1), a corresponding customer id must exist in the *Customer* table. More formally: Let $P_n\{PK, FK\}$ be the parent table and $C_n\{PK\}$ be the child table and $P_n\{FK\}$ is the foreign key referencing $C_n\{PK\}$, then the following must be true: $\forall P\{FK\}\, \exists C\{PK\} : P\{FK\} = C\{PK\}$

Obviously, the same is true for redundant data. Redundancy is regularly employed to increase data processing efficiency. Assume the *OrderLine* fact table, in addition to the customer id, also stores the customer's first and last names. During the generation of data for *OrderLine* valid names from the *Customer* table need to be generated.

Daily Quantity		
OrderDate	CustID	Quantity
2011-03-31	1	350
2011-03-31	2	150
2011-04-01	1	15

OrderLine			
…	Quantity	OrderDate	CustomerID
…	70	2011-03-31	2
…	65	2011-03-31	2
…	15	2011-03-31	2

Figure 6: Aggregation example

Another form of inter table dependency is the generation of data for auxiliary data structures, such as materialized views. For performance reasons one might, for example, store a materialized view containing daily quantities by customer, as seen in Figure 6. This is an aggregation of all quantities ordered by customers. In contrast to the referential integrity, for aggregations a single value in one relation depends on multiple values in another relation.

5. DETERMINISTIC PARALLEL DATA GENERATION

Based on the parallel pseudo random number generation concept presented in Section 3 some of the data with the dependencies described Section 4 can already be generated with PDGF. In order to achieve an acceptable generation speed, the deterministic behavior of random number generators is exploited. As explained in Section 3, each single independent column value of a relation can be generated by mapping its random number or sequence of random numbers to its concrete value. Please note that not all data dependencies have been implemented yet in PDGF. We will note this where necessary in the next sections.

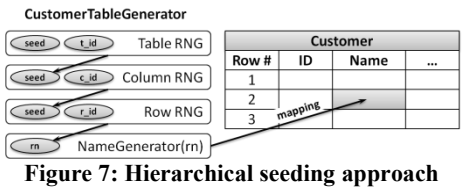

Figure 7: Hierarchical seeding approach

In [11] Rabl et al. presented a hierarchical seeding approach for parallel random number generators that allows for the generation of data with different dependencies. Based on an initial seed for the complete schema seeds are generated in a hierarchical fashion for each table and column in a schema. This way deterministic seeds for every column can be calculated. Based on a column's seed an independent sequence of random numbers is generated to produce its actual values. That is, to produce the *n*-th value in the *m*-th column, the *n*-th random number of the *m*-th column generator is used. This is depicted in Figure 7.

Based on this seeding approach, a general purpose parallel data generation framework (PDGF) has been implemented at the University of Passau. It uses a parallel random number generator to generate data as depicted above. Even for large database schemas with hundreds of tables and thousands of columns PDGF can cache the seeds of all columns. The default random number generator has a period of $2^{64} - 1$ which is enough to generate petabytes of data. The metadata, also referred to as data generators, for a specific database schema is presented to PDGF in form of an XML document and passed to it as an input file. It consists of a hierarchy of generators, following abbreviated with Gen: $SchemaGen \rightarrow TableGen \rightarrow ColumnGen \rightarrow GenericGen$.
Below find the definition of a *IDGenerator* for the *Customer* relation introduced in Section 4. See [11] for more detailed examples.

```
<schema name="warehouse">
 <scaleFactor name="custscale">5000</scaleFactor>
 <seed>1234567890</seed>
 <rng name="PdgfDefaultRandom" />
 <tables>
  <table name="Customer">
   <size>custscale</size>
   <fields>
    <field name="ID">
    <type>java.sql.Types.BIGINT</type>
    <generator name="IdGenerator" />
   </field> […]
```

Figure 8: Sample CustomerGenerator using PDGF syntax

The specification of a schema generator follows the relational schema closely. Each column generator specifies the data type of the field in the relational schema. In the example above, the *IDGenerator* is of type `java.sql.Types.BIGINT` and uses the generic *IDGenerator* implemented as a Java class. The amount of data to be generated is specified in terms of scale factors (see CustScale above). To generate the data in parallel, each relation is partitioned horizontally between the generating instances. This is depicted in the figure below.

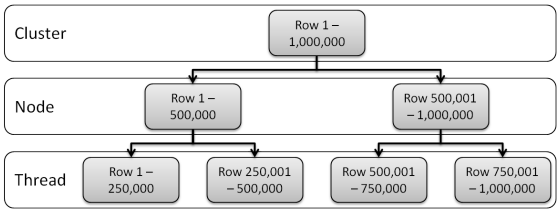

Figure 9: Partitioning hierarchy of PDGF

For instance, if a table with 1,000,000 rows has to be generated on two nodes with four threads/processes, each thread/process generates a continuous partition of 250,000 rows. Since each row is generated independently, the generation speed scales linearly.

Obviously, this form of data generation works well for uncorrelated data. Generating dependant data as discussed in the previous section is more complicated. In the remainder of this section, we clarify what kind of dependencies can be generated with the current version of PDGF. For those dependencies that cannot be implemented with the current version of PDGF we discuss ideas how the future version of PDGF could generated them.

5.1 Intra Row Dependency

Although PDGF generates all fields of a row independently, it is possible to generate all kinds of intra row dependencies. If two columns contain dependent data, as in the case of the date table in Section 4.1, they must share the same random sequence. DateStamp is generated using a generic date generator. For each of the dependent fields, the same generator is invoked with the same random number, but different options reflecting the various date formats. It is also possible, to use a reference or exact value of one or more fields as the basis for value generation. In PDGF a reference is de-referenced by using the deterministic seeding approach to re-compute the value of an arbitrary field. In order to apply the approach above, the generation of the dependent value has to reference either the random numbers of all referenced fields or the values they generate. A reference is defined in PDGF as follows:

```
<generator name="DefaultReferenceGenerator">
 <sameRowAs>ID</sameRowAs>
</generator>
```

The default reference generator re-computes a value of the referenced field. This can be used as input for another generator. An example for the use of the same random number sequence is the VAT example. The list of possible locations for a purchase is stored in a dictionary table *locations* and the concrete choice could be made based on a random number or the value of the location field. To get the VAT for a specific location, a second dictionary table with VAT values is employed, e.g. *VAT*. If *VAT* has the same order as *locations,* the matching VAT can be retrieved using the same random number as used to generate the value for location. Otherwise the VAT value can be picked using the value of the location field and a function mapping location to VATs.

In summary, PDGF has three options to generate intra row dependencies: shared generators, equal random number sequences and referencing fields.

5.2 Intra Table Dependency

In PDGF intra table dependencies are not implemented yet. They could be handled like intra row dependencies. However, intra table dependencies often have a recursive structure and, therefore, do not allow an independent computation of single values. This conflicts with the stateless generation approach of the current version of PDGF. In the following paragraphs we explain how intra table dependencies can be generated in general and PDGF specifically.

Let's assume that each order *ID* in the *OrderLine* example from Section 4 has to appear at least *min* and at most *max* times. That is each order has between *min* and *max Item* entries. In the serial data generation case, the approach is straight forward by generating between *min* and *max* rows at a time using the same order *ID*. However, the parallel case is more complicated, as each thread/process generates data for different *ID*s independently. In a naïve approach this will ultimately lead to a non-deterministic number of *OrderLine* rows per *ID*. This is the state of the art in current parallel data generators. A downside of this approach is that it does not allow an independent computation of a single row. Relational theory does not enforce the sorting of tuples. However, real world data sets exhibit some inherent sorting as could be seen in the examples in Section 4.2. Therefore, the data generation must be able to produce sorted data where necessary. If the sorting has a regular structure, it is usually possible to calculate a value based on its position, without knowing its predecessors. A trivial example for this is a surrogate key.

To generate data with an irregular sorting, in general a sequential approach is needed, since each value is dependent on the previously generated value. Consider time stamps that have to be

increasing with random intervals and each time stamp may occur multiple times. In order to generate a single time stamp knowledge of the previous value is needed. In PDGF, this would lead to a recursive dereferencing to the first value. Obviously, this approach is not feasible for large data sets. Therefore, the generation is usually split into several continuous partitions, which are each generated sequentially by a separate process. This is possible, if some deviations at the partition boarders are acceptable. However, using this approach the resulting data is dependent on the degree of parallelism, which is in contrast to the general requirement of platform independence. A possible solution to generate such data in parallel is to restrict the randomness in the data, i.e. make the sorting dependent of the row number. This can be done by applying a certain reoccurring pattern. Obviously, the data will exhibit the desired pattern.

We are currently exploring an approach using conditional probabilities to generate increasing number sequences with random characteristics. This can be seen as an integral of a random sequence. If it is possible to calculate the integral in constant time, the generation of sorted data will be possible in PDGF.

To generate multi-valued functional dependencies in data, multiple rows have to be generated at a time. This approach can be used in the example above. However, if a continuous surrogate key has to be generated for each row, the problem is similar to the generation of irregular sorted data. A random number sequence is needed, that returns a certain random number multiple times.

5.3 Inter Table Dependency

To ensure referential integrity most data generators either have severe restrictions on the key generation or read generated keys to generate data consistent across tables. Using PDGF's deterministic approach, referential integrity can be ensured by regenerating valid keys. This is equal to the reference generation for intra row dependencies. The row number of each value is used as a surrogate key. To generate a valid key, the random sequence of the key column and the key generator is needed. By picking a random row of the key column the corresponding key is generated. In order to generate statistic distributions of references the random picking of row numbers can be distributed accordingly. This is depicted in Figure 10; to generate a referenced value in the first step a random lognormal distributed value within the number of rows of the referenced table is generated. This row number is used to regenerate the key value. Obviously, it is possible to generate any reference, not just foreign keys.

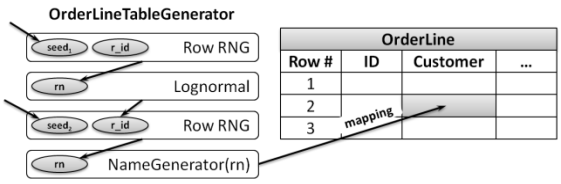

Figure 10: Reference generation in PDGF

A more complex problem is the generation of values that depend on multiple other values. Usually, there is a connection between the referenced values that has to be considered. Although it is possible to use multiple random references within a field and aggregate the according data, this method does generally not lead to the desired result. Consider the example of a materialized view. In this case, one value is dependent on multiple values in another table. Obviously, the reference approach does not work directly for aggregations. However, if the data is grouped by a field which allows a computation of the row number, then the aggregated values can be computed. Consider a daily aggregation of orders; if the time stamp is regular and the start value is known, then it is possible to calculate the first and last row of each day and compute all values in between. Obviously, the computational complexity is proportional to the number of aggregated values.

6. CONCLUSION

Advances in database technology call for new approaches in benchmarking. In the age of cloud computing and social media, new methods for parallel data generation are needed. In this paper, we developed requirements to data characteristics frequent in relational models that have currently not been addressed by generally available parallel data generators. Furthermore, we showed which of these requirements are fulfilled in PDGF. Some dependencies cannot be generated with PDGF. In future work, we concentrate on the generation of these unsolved dependencies in PDGF.

7. REFERENCES

[1] N. Bruno and S. Chaudhuri. 2005. Flexible database generators. *VLDB '05*, 1097-1107. VLDB Endowment.

[2] P. D. Coddington. 1996. Random number generators for parallel computers. *NHSE Review.*

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with YCSB. *SoCC '10*, 43-154. ACM.

[4] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger.1994. Quickly generating billion-record synthetic databases. *SIGMOD '94*, 243-252. ACM.

[5] J. E. Hoag and C. W. Thompson. 2007. A parallel general-purpose synthetic data generator. *SIGMOD Record*, 36(1):19-24.

[6] K. Houkjær, K. Torp, and R. Wind. 2006. Simple and realistic data generation. *VLDB '06*, 1243-1246. VLDB Endowment.

[7] P. J. Lin, B. Samadi, A. Cipolone, D. R. Jeske, S. Cox, C. Rendón, D. Holt, and R. Xiao. 2006. Development of a synthetic data set generator for building and testing information discovery systems. *ITNG '06*, 707-712. IEEE Computer Society.

[8] G. Marsaglia. 1991. Normal (Gaussian) random variables for supercomputers. *Journal of Supercomputing,* 5 (1991):49-55. Kluwer Academic Publishers.

[9] G. Marsaglia. *The Diehard Battery of Tests of Randomness*. http://www.stat.fsu.edu/pub/diehard/

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery *Numerical Recipes – The Art of Scientific Computing*. 2007. Cambridge University Press.

[11] T. Rabl, M. Frank, H. Mousselly Sergieh, H. Kosch. 2010. A Data Generator for Cloud-Scale Benchmarking. *TPC TC '10*, 41-56. LNCS 6417. Springer.

[12] J. M. Stephens and M. Poess. 2004. MUDD: a multi-dimensional data generator. *WOSP '04*, 104-109. ACM.

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

[14] M. Stonebraker. 2009. A New Direction for TPC? *TPC TC '09,* 11-17. LNCS 5895. Springer.

[15] A. Thusoo,J. S. Sarma,N. Jain,Z. Shao,P. Chakka,N. Zhang,S. Anthony,H. Liu,R. Murthy: Hive - a petabyte scale data warehouse using Hadoop. ICDE 2010: 996-1005