Efficient Update Data Generation for DBMS Benchmarks

Michael Frank
University of Passau
Faculty of Computer Science
and Mathematics
Passau, Germany
frank@fim.uni-passau.de

Meikel Poess
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA-94404
meikel.poess@oracle.com

Tilmann Rabl
University of Toronto
Department of Electrical and
Computer Engineering
Toronto, ON, Canada
tilmann.rabl@utoronto.ca

ABSTRACT

It is without doubt that industry standard benchmarks have been proven to be crucial to the innovation and productivity of the computing industry. They are important to the fair and standardized assessment of performance across different vendors, different system versions from the same vendor and across different architectures. Good benchmarks are even meant to drive industry and technology forward. Since at some point, after all reasonable advances have been made using a particular benchmark even good benchmarks become obsolete over time. This is why standard consortia periodically overhaul their existing benchmarks or develop new benchmarks. An extremely time and resource consuming task in the creation of new benchmarks is the development of benchmark generators, especially because benchmarks tend to become more and more complex. The first version of the Parallel Data Generation Framework (PDGF), a generic data generator, was capable of generating data for the initial load of arbitrary relational schemas. It was, however, not able to generate data for the actual workload, i.e. input data for transactions (insert, delete and update), incremental load etc., mainly because it did not understand the notion of updates. Updates are data changes that occur over time, e.g. a customer changes address, switches job, gets married or has children. Many benchmarks, need to reflect these changes during their workloads. In this paper we present PDGF Version 2, which contains extensions enabling the generation of update data.

Categories and Subject Descriptors

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

General Terms

Measurement

ICPE'12, April 22–25, 2012, Boston, Massachusetts, USA

Keywords

Benchmark, Data Generation

1. INTRODUCTION

Over the years industry standard benchmarks have proven to be crucial to the innovation and productivity of the computing industry. They are important to the fair and standardized assessment of performance across different vendors, different system versions from the same vendor and across different architectures. Hence, they are used by system and software vendors as much as they are used by customers during their purchase decisions. Vendors use them to differentiate their systems‘ performance from those of other vendors, while they use them internally to monitor systems‘ performance across releases and, in case of software, even across daily builds. Customers use industry standard benchmarks to cost effectively compare the performance of systems. Without benchmarks they would need to invest in the performance analysis of different systems.

In the last 25 years many industry standard benchmarks were developed by benchmark consortia. Among these consortia are, most notably, the Standard Performance Evaluation Corporation (SPEC), the Transaction Processing Performance Council (TPC) and the Storage Performance Council (SPC). While focusing on slightly different aspects of system performance, they all follow a common goal, namely to develop fair and verifiable standard benchmarks, and to police and publish results on a large number of systems. In this paper we focus on those benchmarks that measure performance of database management systems (DBMS).

While the fundamental steps in a DBMS benchmark have remained the same over the years, namely to load the initial database and then to run a workload, e.g. transactions or queries, the complexities of the benchmarks have increased enormously [11]. For instance, TPC-A, TPC's first OLTP benchmark specification, counted 43 pages and used a single, simple, update-intensive transaction to emulate update-intensive database environments. The transaction access a schema with four tables, all with 1:n relationships. In comparison TPC-E, TPC's latest OLTP benchmark specification counts 286 pages and consists of 10 transactions emulating the complex operation of a brokerage firm. Its transactions access a schema with 9 tables and complex relationships.

Each benchmark requires tools to generate datasets for the initial load and the subsequent workload. In [21] we have analyzed data generation requirement of today's complex DBMS environments. While the data dependencies in

TPC-A were restricted to foreign-key/primary key relationships and data volumes were in the gigabyes, TPC-E's data dependencies are more complex. For the initial load of a DBMS we have identified three classes of data dependencies: i) intra row dependencies, e.g. the city and zip code of an address table, ii) intra table dependencies, e.g. the n:1 relation-ships of normalized schemas and iii) inter table dependencies, e.g. Foreign key, primary key relationships. In [21] we have shown how the Parallel Data Generation Framework (PDGF) can be configured to generate the above data.

Since then we have been working on PDGF based data generators for TPC-H and a new extract, transform and load (ETL) benchmark [25] of the Transaction Processing Performance Council (TPC). This work has revealed that the above data dependencies also exists in the data sets. Consider the two tables of a customer management system:

$$Customer = \{C_1, C_2, ..., C_n\} \quad (1)$$

$$CustomerAddress = \{CA_1, CA_2, ..., CA_n\} \quad (2)$$

with the primary keys C_1 and CA_1 and the foreign key CA_2 to join to C_1 of the customer table. Now consider the following workload: For a random customer find his address and update each field C_i with a likelihood of L_i. In case the customer address key range is dense and static finding an existing customer and his address is trivial. But what if the keys are not dense, or we add customers as part of the workload. Then the key range evolves over time making it more difficult to pick valid keys, especially in parallel data generation.

The above scenario is very common in ETL systems as their main workload is to load data, especially incremental data that can contain updates to pre-existing data. The ETL benchmark, currently under development by the TPC, defines two sets of schemas, one to emulate the source tables and one to define the target tables. The workload is defined in terms of transformation that map the input data of the source tables to the output data of the target tables. As a result writing a data generator for an ETL benchmark is very challenging. The update data needs to be generated deterministically, i.e. for each table one needs to be able to set the number of new, updated and deleted records.

In this paper we present our extensions to PDGF that enable the deterministic generation of intra-table dependencies and updates for database benchmarking, which were developed specifically to support TPC's ETL benchmark development. Our main contributions are:

- the principle of a growing permutation, that is basis for our update generation strategy,
- an extension of our data generation approach to generate intra-table dependencies,
- the development of a framework for transparent, consistent and repeatable generation of inserts, updates, and deletes for generated data.

The remainder of this paper is organized as follows: In the next section we discuss related work, especially data generators that try to solve similar problems. In Section 3 we briefly review the architecture and design goals of the PDGF. Section 4, the main contribution of this paper, details the basic principles of our approach for generating updates and how it is integrated in PDGF. We present some performance numbers in Section 5, before concluding in Section 6.

2. RELATED WORK

Data generation for performance evaluation is part of the daily business of researchers and DB administrators. Most of their data generators are special purpose implementations for a single dataset. The active demand for generic data generation tools feeds a lively industry in this niche that sells their generators for hundreds to thousands of dollars. Examples of commercial tools are Red Gate SQL Data Generator [22], DTM Data Generator [6], and GS DataGenerator [8]. But still new companies are entering this segment, e.g. LogicBlox with their TestBlox software.

There has been quite a lot of research on data generation for performance benchmarking purposes. An important milestone was the paper by Gray et al. [7], the authors showed how to generate data sets with different distributions and dense unique sequences in linear time and in parallel. Fast, parallel generation of data with special distribution characteristics is the foundation of our data generation approach.

Most scientific data generators have been built for a single benchmark as well. Examples of simple data sets (i.e. that only consists of single tables or or unrelated tables) are SetQuery [15], TeraSort, MalGen [1], YCSB [5], the Wisconsin database [2], and Bristlecone [4]. To generate data intra- and inter-table dependencies – as specified in [21] – the data generators have to be more sophisticated. For large data sets two solutions are common: either to re-read the generated data or to use a per user simulation for the generation. An examples for the former approach is dbgen [24], the data generator provided by the TPC for the TPC-H benchmark[17], another approach was presented by Bruno and Chaudhuri [3]: it largely relies on scanning a given database to generate various distributions and interdependencies. Two further tools that offer similar capabilities are MUDD [23] and PSDG [9]. Both feature description languages for the definition of the data layout and advanced distributions. The later approach is often realized by graph based models as presented by Houkjær et al. [10] and Lin et al. [12].

All of these data generators consider the benchmarking as a two phased procedure, first the data has to be loaded and then a workload must be processed. Therefore, they generate a historical load and rely on a workload generator for queries and updates. However, in some cases the historical workload also reflects the stream of updates, e.g. in an history-keeping dimension [14]. This cannot be generated by either of the generators above.

3. PARALLEL DATA GENERATION FRAMEWORK

The Parallel Data Generation Framework (PDGF), developed at the University of Passau, is a generic data generator for relational data [19]. It was designed to take advantage of today's multi-core processors and large clusters of computers to generate exabytes of synthetic data very quickly. It was originally built for the generation of large, relational data sets stored as flat files. Since the release of PDGF Version 1.0 major parts have been extended and many new functionality has been added.

The most important improvement in *PDGF 2.0* is the

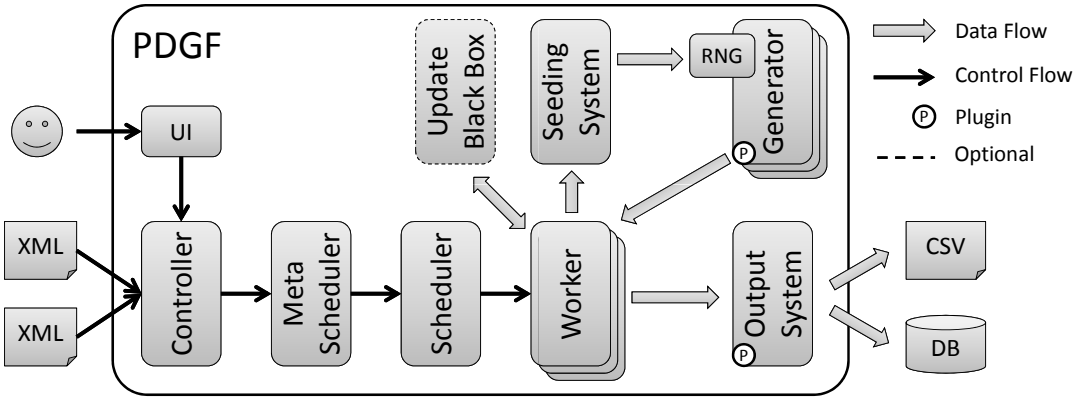

Figure 1: PDGF Architecture

ability to represent and generate data changes over time. *PDGF 2.0* is capable of performing multiple updates throughout a benchmark run, like adding, changing and deleting tuples. This allows the simulation of the natural growth of a data set over time. Changed data can be generated as a changed data capture file (CDC file), containing all the transactions and changes in a specific time interval, or as a snapshot of the entire table at a specific point in time. The high data generation speed is achieved by exploiting multi-core processors running in large clusters using the parallelism in pseudo random numbers. PDGF uses a fully computational approach and is a pure Java implementation which makes it very portable.

At its core PDGF consists of a set of functions that maps virtual IDs to row values using hierarchically seeded random number generators. Currently, PDGF uses 8 Byte integers for seeds and random numbers.

Each table, each column of a table and each row of a table have their own deterministic seeds, seeding a corresponding random number generator with a period of $2^{64} - 1$, long enough to generate petabytes of data. The seeded random number generators are then used to compute the row values. This unique seeding approach enables PDGF to quickly generate any row value for each field of a table independently and deterministically. Even for large relational schemas with hundreds of tables, and thousands of columns per table the total number of seeds for tables and columns is manageable and can be cached in PDGF. For instance, 1000 tables with 1000 columns each requires a cache of only 7.6 Megabytes. 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. We extended the seeding strategy to reflect time-related dependencies and will give further details in Section 4.1.

Since the first publication of PDGF we have added many features and replaced or extended about 80% of its code base. The result, *PDGF 2.0*, will be presented in detail in the following section.

3.1 Architectural Overview

PDGF is designed as a generic data generator for benchmarking relational database systems. It was built with the intention to enable a fast generation of non-trivial datasets as used in TPC-H [18]. As such PDGF was designed with the following four goals in mind:

- Configurability: PDGF is configurable to generate any type of schema. The description of the data schema and the generation output are described by two separate XML files.
- Extensibility: PDGF can be extended to allow for the implementation of future requirements. Nearly every aspect of PDGF is exchangeable and expandable through plugins, enabling a broad band of applications.
- Scalability: This is achieved by parallelizing data generation in threads across processor cores, processors and machines without many thread dependencies to avoid costly inter-thread communication.
- Efficiency: PDGF efficiently uses all available system resources while scaling linearly.

Figure 1 presents a high-level overview of PDGF's architecture. The controller, depicted on the left side, is the interface to the user by means of input files and a user interface (UI), which can be either a graphical user interface, a interactive shell, or a command line interface. The controller reads meta-data about the schema to be generated, i.e. table definitions, value distributions, output formats and system configurations, from two XML files and initiates the meta scheduler. The meta scheduler organizes the data generation across multiple machines. It also instantiates the scheduler that spawns multiple worker threads (by default one for each core). The scheduler divides the work and assigns equal sized, continuous portions of the data to each worker. The actual data generation is done by so called generators, which are executed in the worker threads. The workers use the seeding system to give a correctly instantiated random number generator to the data generators. These generate the actual values. To generate non-uniform data the system features various distributions that can be applied to the random numbers. If updates have to be generated, the worker use the optional update black box to retrieve the correct update ID (see section 4.6). The values are then written to the output system that writes them to files or any other target. The output system can further manipulate the data, e.g. splitting or grouping the generated data into multiple files

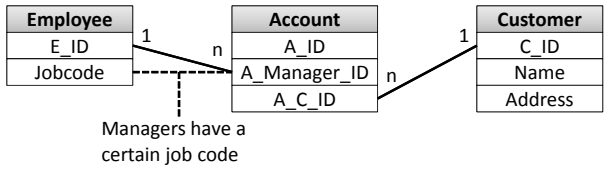

Figure 2: Example schema

and apply further formatting. As indicated, data generators and output modules can be replaced or extended by user defined plugins. The workflow of a data generation process is as follows:

- Initialization: The controller, guided by the user, loads and parses the configuration files to determine the schema and value distributions, and instantiates the schedulers.
- Partitioning: The meta scheduler determines, which data has to be generated by a particular PDGF instance. The scheduler partitions the workload further and assigns work units to worker threads,
- Data Generation: For each column the worker invokes the appropriate data generator that generates the actual value and forwards the data to the output.
- Data Output: PDGF supports many different output formats. The output system formats the generated data as specified in the configuration and writes it to the specified target, e.g. a file.

The following sections discuss our approaches to meet our design goals, i.e. configurability, extensibility, scalability, and efficiency.

3.2 Configurability

PDGF can be controlled by command line parameters, via the built-in interactive shell or through external components like a GUI. The data generation in PDGF 2.0 is controlled by two XML based configuration files: i) the *Schema XML* file and ii) the *Generation XML* file.

The schema XML file describes the schema of the data set to be generated, how values for a certain table columns should be generated, the relationship between table columns and the distributions of table values.

Listing 1 shows the schema configuration file for the schema introduced in Figure 2. It starts with the definition of three properties `prop`, i.e. SF, Size_Emp and SizeAcc. Support for multiple properties was added in PDGF 2.0 for an easier adaption of table sizes and in order to express non-linear growth between tables. However, properties can be deployed to parameterize the configuration. After the properties are defined the project seed is set. This seed is the *root* seed. By changing it the complete data set will change. After the seed the default random number generator is specified. It is possible to specify different random number generator implementations for different parts of the generated data. The definition of the schema follows. It defines three relations: employee, account and customer. Each relation is represented by a `table` tag. Within each table tag single attributes are defined using `field` tags. They define generators, distributions and references.

```
1  <schema name="mySchema">
2  <prop name="SF"         type="long">10</prop>
3  <prop name="Size_Emp"   type="long">50</prop>
4  <prop name="Size_Acc"   type="long">200*SF</prop>
5  <seed>1234567890</seed>
6  <rng name="PdgfDefaultRandom"/>
7  <table name="employee">
8   <size>SF * Size_Emp</size>
9   <field name="e_id" type="INTEGER">
10   <generator name="pdgf.generator.IdGenerator"/>
11  </field>
12  <field name="jobcode" type="INTEGER">
13   <size>20</size>
14   <generator name="pdgf.generator.PermuteJobID"/>
15  </field>
16 </table>
17 <table name="account">
18  <size>Size_Acc</size>
19  <field name="a_id" type="INTEGER">
20   <generator name="IdGenerator"/>
21  </field>
22  <field name="a_manager_id" type="INTEGER">
23   <generator name="pdgf.generator.PermuteJobID">
24    <reference>
25     <table>employee</table><field>jobcode</field>
26    </reference>
27   </generator>
28  </field>
29  <field name="a_c_id" type="INTEGER">
30   <generator name="DefaultReferenceGenerator">
31    <reference>
32     <table>customer</table><field>cust_id</field>
33    </reference>
34   </generator>
35  </field>
36 </table>
37 <table name="customer" type="update">
38  <size>SF*100</size> <!--initial table size-->
39  <newPercentage>20</newPercentage>
40  <updatePercentage>75</updatePercentage>
41  <deletePercentage>5</deletePercentage>
42  <!--size of each update batch-->
43  <UpdateSize>50 * SF</UpdateSize>
44  <updateFirstID>1</updateFirstID>
45  <updateLastID>3</updateLastID>
46  <field name="c_id" type="INTEGER">
47   <updatePercentage>0</updatePercentage>
48   <generator name="pdgf.generator.IdGenerator"/>
49  </field>
50  <field name="name" type="INTEGER">
51   <updatePercentage>0</updatePercentage>
52   <generator name="pdgf.generator.DictList">
53    <file>dicts/Given-Names.dict</file>
54   </generator>
55  </field>
56  <field name="address" type="INTEGER">
57   <updatePercentage>0.25 * 100</updatePercentage>
58   ...</field>
59 </table>
60 </schema>
```

Listing 1: Schema Configuration File

To enable the specification of complex relations between tables and virtual tables, we have added a second XML configuration file, the Generation XML file. Virtual tables are not generated, but used for referencing instead. The generation XML file defines how data structures are defined, what scheduler strategy to use, and how to perform the final processing before the generated data is either directly stored in a database, written to flat files or XML files. See Listing 2 for an example of a Generation XML file for the example of Figure 2. Flat files can be specified using a template, which is in-line Java code and compiled at runtime. For examples see Listing 2 line 9 and 20.

```
<project>
 <scheduler name="DefaultScheduler"></scheduler>
 <output name="CSVRowOutput">
  <sortByRowID>true</sortByRowID>
  <delimiter>|</delimiter><!-file field separator>
  <outputDir>output/</outputDir>
  <fileEnding>.txt</fileEnding>
  <fileTemplate>table.getName() +
      fileEnding</fileTemplate>
 </output>
 <schema name="mySchema">
  <table name="account">
   <scheduler name="UpdateScheduler"/>
   <output name="CSVRowOutput">
        <sortByRowID>true</sortByRowID>
    <delimiter>|</delimiter>
    <outputDir>output/</outputDir>
    <fileEnding>.txt</fileEnding>
    <fileTemplate>"Batch"+(updateID+1)
    +"/"+table.getName()+fileEnding</fileTemplate>
   </output>
  </table>
 </schema>
</project>
```

Listing 2: Generation Configuration File

```
public class MyOwnGenerator extends Generator {
 private String myOwnNodeValue;
 boolean required=false, used=true;
 public MyOwnGenerator() {
  super("My first generator");
 }

 public void nextValue(AbstractPDGFRandom rng,
      GenerationContext gc,
   FieldValueDTO fv) {
  fv.setValue(myOwnNodeValue);
 }

 protected void initStage0_configParsers(){
  addNodeParser(new Parser(required, used,
       "myOwnNode", this, "someDescription" ) {
   protected void parse(Node node){
    myOwnNodeValue=node.getTextContent();
    if(myOwnNodeValue.isEmpty())
     error("must not be empty!");
   }});
}}
```

Listing 3: Example of Simple Generator Implementation

PDGF's template approach is very flexible. PDGF 2.0 features a *TemplateOutput* plugin, allowing the specification of the exact formating of generated information within the XML file using plain Java code. Despite the mixture of XML and Java code this approach has many benefits for the configuration file writers. They do not require a Java SDK or an integrated development environment (IDE) to be installed to create a new or adapt an output plugin. This is especially useful if PDGF is running on a server and is controlled remotely using e.g. a terminal session. Using the *TemplateOutput*, PDGFs output formating can be quickly and easily adapted by using any available text editor. During runtime the Java code is extracted from XML, compiled and loaded entirely in memory using the Javassist framework, which is itself written entirely in Java. As real bytecode is generated, there is no performance penalty using this approach instead of writing a dedicated output plugin class. The Java in XML template concept can also be applied to generators, speeding up their testing and development. However, writing a dedicated plugin class is still the preferred way of extending PDGF, as such a class can be reused and they offer better configurability and better access to PDGF's internal APIs.

Both XML files are validated by the framework on a modular basis. This means that XML subtrees are parsed directly by the modules that need the information. Plugin writers can easily extend the XML node parser for their components. An example can be seen in in Listing 3.

PDGF's internal seeding strategy, parallelization, and reference generation is encapsulated in what we refer to as a black box. The plugin API and the configuration files hide the internals and make it therefore relatively simple to write a custom plugin. While PDGF abstracts from parallelization details as much as possible for certain plugin types, some parallelization details have to be considered. This is for example the case for scheduling and output plugins since these are synchronization points within the framework.

3.3 Extensibility

Since we aimed to built a generic data generator, extensibility is a paramount design goal. Therefore, our implementation allows for all major parts of PDGF to be extended and replaced by custom plugins. For each module a corresponding superclass within the pdgf.plugin package exists. To implement a custom plugin this superclass can be extended. PDGF automatically imports all plugins that are stored in its plugin folder. Alternatively, plugins can be loaded with command line arguments and during runtime. To use a custom plugin in the schema configuration file, its fully qualified class name needs to be specified. This can be seen in Listing 1 line 10.

Every plugin is part of a parent/child(s) structure, which resembles the XML Document Object Model tree. This makes accessing and navigating the internal data structure very easy and intuitive. A properties API allows for variable assignment and simple arithmetic calculations within the XML file. This can be seen in Listing 1. PDGF supports long, double and date values, non-nested brackets and the four basic operations $\{+, -, *, /\}$. Properties can be nested as shown in Listing 1, where the value of property *Size_Acc* is dependent of property *SF* multiplied by 200.

3.4 Scalability

Traditionally the generation of synthetic data scales very nicely as long as there are no synchronization points between processes/threads. Hence, the goal for PDGF was to reduce synchronization between processes/threads to an absolute minimum. However, some synchronization is still required. The most notable synchronization points are those to shared resources like the hard disk. All synchronization critical data structures are cloned for each thread. PDGF's computational approach of data generation in general provides for a synchronization, cache and disk/network read access free way of generating data. PDGF does not rely on an underlying database management system or other caching facilities to store intermediate values or resolving data interdependencies. This allows linear scalability across processor cores and systems, as every worker thread and instance of PDGF generates its own data partition without the need of waiting for intermediate results of other parts or control communication between them. This is especially important in shared

nothing architectures where the exchange of data between nodes is very expensive.

3.5 Efficiency

The same strategies that enable the scalability are responsible for the efficiency of the framework. Every complex data structure is an organization of the relations between its entities. To generate a relation, one must know the value of the related entity. With an computational approach disk and network access can be avoided by computing values. When comparing the amount of CPU cycles required e.g. for hard disk access to the amount of cycles required to (re-)calculate the value on demand in nearly all cases the (re-)calculation is much cheaper. The same is true for in memory cache structures. The costs for maintaining e.g. a LRU cache data structure outweighed its benefits of avoiding (re-)calculation. This may not be true for every data set one wishes to generated, but for most cases this is true. The only value caching strategy of PDGF is to cache the last generated value of each generator for intra tuple dependencies. PDGF's ability to utilize all available compute resources and avoiding idle times due to synchronization are providing it with an outstanding efficiency.

4. COMPUTATIONAL APPROACH TO DATA GENERATION

As long as there are no dependencies between data values, PDGF can compute each data value completely independent. Even in the presence of data dependencies there is no need to scan for values or cache referenced values, as each value can be recalculated by reseeding the random number generators. The underlying principle is a series of mapping steps. Each data value in the data set has a virtual unique key similar to an virtual address in a database system. With this key the data value can be addressed. The addressing in a table can be done the same way as in a 2 dimensional matrix, each cell can be identified with the table's columns and rows. The complete key K of a value consists of three identifiers:

$$K : (\textit{table ID, column ID, row ID})$$

To track changes of the values over time PDGF 2.0 uses an additional attribute, the *update ID*. The resulting key consists of four identifiers:

$$K : (\textit{table ID, column ID, update ID, row ID})$$

The update ID addresses an abstract time unit. If the time component is not required the update ID is always set to zero. Otherwise the range of legitimate update IDs is specified in the schema configuration file (an example can be seen in Listing 1, line 45-46). All values addressed by a key are usually derived from a random number that is generated by a permutation or a random number generator (RNG). Both, the permutation and the RNG require an initial value, i.e. a *seed*, to initialize their internal state. PDGFs data generation approach requires the permutation or random number generator to deterministically generate the random number for a given seed and key. Based on this key a value generator deterministically generates the concrete value. That means that a repeated computation of a value will always lead to the same result. Since the values are derived from a random number, the RNG or permutation must always start from the same internal state, i.e. from the same seed. This assures that the generation uses the same stream of random numbers for a repeated generation of the same values. PDGF uses the following steps to deterministically generate a value for a certain key K:

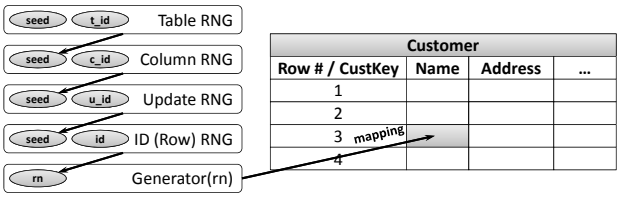

Figure 3: PDGF's Seeding Strategy

1 generate a unique seed S for K, $S \rightarrow K$,

2 seed a random number generator with S

3 pass the seeded RNG to the value generator which is responsible to generate the actual value

4 the generator uses the provided RNG to generate the required amount of random numbers and calculates the concrete value

The seed is generated using the deterministic seeding strategy which will be discussed in detail in the next section. The value generator uses the generated random numbers to deterministically generate a value, a very common generator is a dictionary lookup: the line number is calculated by computing the modulo of the random number. This is also used in the example in Listing 1, line 53 (the `DictList` generator).

4.1 Seeding Strategy

In PDGF the mapping of a key to the corresponding seed is realized by a hierarchical seeding strategy. Basically, multiple random number generators are chained. In Figure 3 this hierarchical organization can be seen. To generate the random number of a certain field in the data set the seed for the relevant field has to be calculated. Starting from a single master seed (see listing 1 line 5) for the complete data set (project) the table RNG generates a series of seeds one for every table. Each table seed seeds a new RNG which is used to generate seeds for every column. With the column seed the update RNG is seeded which in turn seeds the row RNG. The row RNG then generates a seed for every field, which is used to seed the value generator which deterministically generates the final value.

This strategy is very efficient since most of the seeds can easily be cached. The number of tables and columns which is also the number of their seeds is usually relatively small and does not change during the generation process. Therefore, these seeds can be cached. The update ID also changes only periodically and can therefore be cached as well. Only the row changes very frequently. Since the seeds are organized hierarchically it is only necessary to retrieve the lowest cached seed. With this seed the row RNG is seeded. These operations are computationally inexpensive. With the seeded row RNG the random number that is basis for

```
void skip(long step){
        seed += step;
}

long next() {
 ++seed;
 long x = seed;
 x = x ^ (x >>> 15);            //XOR1
 x = x ^ (x << 35);             //XOR1
 x = x ^ (x >>> 4);             //XOR1
 x = 4768777513237032739L * x; //MWCG
 x = x ^ (x << 17);             //XOR2
 x = x ^ (x >>> 31);            //XOR2
 x = x ^ (x << 8);              //XOR2
return x;
}
```

Listing 4: Counter-PRNG Hash Function

the value generation can be generated. A problem arises during reference computation. If a reference must retrieve the value from row 1000, a normal RNG had to be seeded and called 1000 times to retrieve the 1000th random number in the random number stream. This would be to costly. The problem can be solved if the RNG is capable to jump to the n-th random number in the random number stream without calculating all n-1 values before. The function to directly jump to a certain position in the random number stream is called skip ahead. The random number generator algorithm has to directly support this function.

4.2 PRNG

The deterministic generation of data relies largely on the properties of pseudo random number generators. To be more precise, a qualifying random number generator has to satisfy the following properties:

i) It must be very fast, i.e. computationally inexpensive; ii) It must be able to generate random values for all number primitives of Java; iii) It must have a sufficiently long period, to be able to generate petabytes of data; iv) It must be statistically sound in its value distribution; and v) It must support a computationally inexpensive skip ahead function.

Java's default random number generator does not satisfy these requirements because of its short period, poor distribution of values, lack of speed and most of all, the missing skip ahead functionality.

The PRNG used in PDGF generates random numbers by hashing successive counter values where a seed serves as the initial counter value. This approach makes skipping ahead an arbitrary number of values inexpensive as it is only a single add operation of two long primitives. The algorithm itself, as shown in listing 4, is a combined generator consisting of two XOR-shift-generators surrounding a multiply-with-carry-generator. The design of the algorithm was inspired by the works of Marsaglia [13] and Panneton and L'Ecuyer [16].

This above default PRNG can be exchanged with any other PRNG as long as all of the above requirements are met. In addition it is possible to specify a different PRNG for each generator.

4.3 Permutations

Pseudo random numbers are not sufficient to generate all types of complex data. Three examples of data that cannot be generated by a PRNG presented above are:

1 Streams of unique numbers in a predefined range (for example for sampling without replacement),

2 Streams of random numbers with exact distributions (for example generate 100 random values with exactly 20 times value "2" and 80 times with value "6" without a counter), and

3 Inverted random numbers, i.e. generate the position in the stream of a random number.

A PRNG is a injective mapping function, that means it maps distinct elements of its domain to distinct element of its codomain. In general and specifically in the case of our PRNG, this mapping cannot be reversed, so it is not possible to map a random number back to its seed.

Unless all generated random numbers are cached, PRNGs cannot be used for sampling without replacement. However, permutations can be used for sampling without replacement. The simplest way of generating a random permutation of values in a range [0, n] is shuffling an array containing the numbers 1 to n. This approach is time and memory consuming. The later is a big problem if billions of random numbers need to be generated. Allocating a sufficiently big array is inefficient and in some cases even impossible due to memory or programing language restrictions (in Java the maximum array size is $2^{31}-1$. Storing long values in such an array requires 16 gigabyte). A permutation algorithm suitable to be used in a parallel data generation environment must meet the following criteria. The permutation function: i) Must be a computational approach to do the mapping, ii) Must be reasonable fast (computationally inexpensive), iii) Must be randomizeable by means of a seed, and iv) Should be bijective.

A permutation can be used in more complex ways than just for sampling without replacement if the permutation is a bijective function. Consider the following example from listing 1: An employee table E with primary key E_{eID} contains records for all employees of a company. Employee job types are identified by a numerical field $E_{JobCode}$ within this table. An account table A has a foreign key $A_{amanager_id}$ referencing employees in E through their E_{eID}. However, only employees are referenced with a specific job type, e.g. $E_{JobCode}$=315 representing *account managers*. Using the seeding strategy described in 4.1 the $E_{JobCode}$ of an employee can be easily recomputed if the employee's $A_{amanager_id}$ is known. During the generation of data for the *Accounts* table we need to generate employee $A_{amanager_id}$ that correspond only to employees that are account managers. One approach would be to pick a random E_{eID} and see whether this employee is an account manager. If yes, this E_{eID} is picked. If not, a new E_{eID} is picked. Besides being inefficient, this approach is not deterministic, especially in parallel data generation. Another approach is to cache the employee table or generate it completely every time. For large employee tables, this is either very memory intensive or a computational nightmare.

A computational solution is to use a bijective mapping function as shown in Figure 4. If the left side in Figure 4 represents the employee IDs (E_{eID}) then the right side is grouped into blocks of job codes ($E_{JobCode}$). To determine if a specific employee, identified with his ID E_{eID} is an *account manager*, we compute the permutation for the ID in the

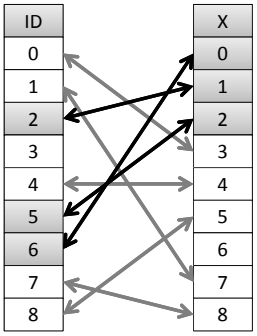

Figure 4: Bijective permutation with offset

following way:

$$X = perm(e_I D) \tag{3}$$

If the permutation result X is within a certain block e.g let the first block [0,2] represent *account managers*, it is assigned the $E_{JobCode}$=315, i.e. the employee job code for account manager. In Figure 4 this is the case if X is within [0,2] on the right side of the permutation. To get the $E_{e_I D}$ for a random *account manager*, we select a random number R from the block of account manages, which is between [0,2] on the right side and compute the reverse permutation:

$$e_I D = inversePerm(R) \tag{4}$$

Let R be 2, then the inverse permutation for the example in Figure 4 is 5. Now we know, that the employee with employee ID ($E_{e_I D} = 5$) is an account manager. This way it is possible to pick (compute) a random employee from the set of account managers without caching, precalculating or storing any other information than the total amount of existing account managers within the employee table. In the next section we will explain how to compute a bijective permutation.

4.4 Bijective Permutation

PDGFs implementation of an bijective permutation is based on a linear permutation polynomial. It fulfills all the above required criteria. It is a special case of quadratic permutation polynomials $g(x) = ax^2+bx+c$ for the ring $\mathbb{Z}/p^k\mathbb{Z}$, with $k = 1$. In the case $k = 1$ a must be 0 and $b \neq 0$. The linear permutation polynomial is therefore defined as $g(x) = bx + c$; $\mathbb{Z}/p\mathbb{Z}$; $b \neq 0$ and the inverse permutation as $g^{-1}(y) = (x - c) \cdot b^{-1}$, where b^{-1} is the inverse element of b in $\mathbb{Z}/p\mathbb{Z}$. This implies that $b^{-1} \cdot b \ mod \ p = 1$. The value for p is the number of elements in the permutation. To define different permutations polynomial we have to determine different values for b b^{-1} and c using a seed similar to a PRNG. In fact our implementation uses the seed to seed a PRNG to generate random values. For c we choose a positive random number. Obviously, p is not necessarily a prime number. If p is not prime, there might not be an inverse element for every possible b in $\mathbb{Z}/p\mathbb{Z}$. Hence, suitable values for b and b^{-1} must be found. To find a suitable b, b is chosen randomly until it satisfies the condition $b^{-1} \cdot b \ mod \ p = 1$. Without such a pair of b's the permutation does not work.

Once valid values are found the algorithm is very simple as can be seen in Listing 5. There are two important implementation details. First, Java's % operator is not a real modulo

```
perm(x){
 y = (x * b + c) mod p
 return y;
}

invperm(y){
 x=((y - c) * b_inv) mod p
 return x;
}
```

Listing 5: Bijective permutation

but a remainder and cannot be used. Second, the limited range of values: as PDGF is intended to generate billions of rows/ID, the multiplication of x and b using long primitive may overflow resulting in a erroneous calculation. Our implementation, therefore, tests if the values could overflow and if true uses `BigIntegers` instead of primitives supporting numbers greater than `Long.MAX_VALUE`.

4.5 Growing Offset Permutation

The simple permutations described in Section 4.3 work if the number of elements is static. However, if the number of elements varies as, for example for updates in a table, a more involved approach is needed. Consider the following variation of the example illustrated in Section 4.3: Let's assume we want to add or remove entries in the customer table from listing 1 due to common actions such as gaining new customers, removing old customers. These actions change the number of elements in the permutation which is not supported by a general permutation.

To solve this problem we have developed a permutation with flexible number of elements, i.e. a permutation with a mechanism to add and remove elements without destroying the existing permutation. We call this permutation *growing offset permutation*. The name indicates that our permutation can grow and be reduced by a certain offset. The offset concept is the same as used in Section 4.3 to identify employees which are *account managers*. We define a mapping of ID's on one side to continuous groups of offsets on the other side of the permutation. The offsets are used to determine the amount of IDs added and removed from the permutation. Basis of the growing offset permutation are multiple instances of bijective permutations. The instances are organized in a hierarchical structure as shown in Figure 5.

There are practical limitations to the depth of such a hierarchy of bijective permutations. To keep the size manageable we introduced the notion of an abstract time unit T, which we call **generation**. During every generation elements can be removed and added. If the likelihood of removing elements from the permutation is higher than the likelihood of adding elements the growing offset permutation eventually runs out of elements, i.e. we will have an empty generation against which no updates are possible. Otherwise if the likelihood of adding an element is higher the number of elements will grow. If both, adding and removing, have equal probability the size will be constant. A generation T can be seen as a snapshot containing the state of the table during the time period of length T. The period of a generations can be specified, e.g. seconds, minutes, days or years, depending on the desired granularity.

Each generation requires its own instance of a bijective

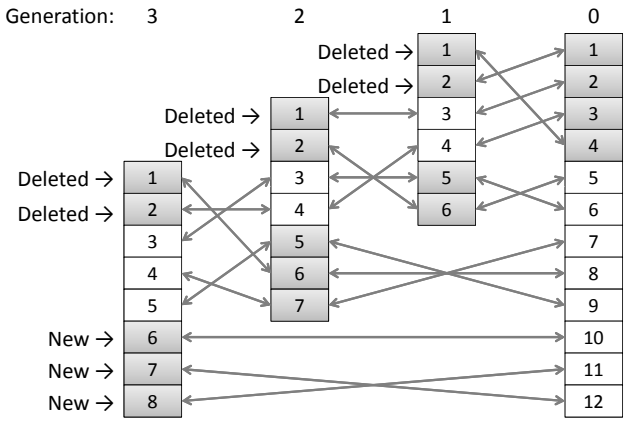

Figure 5: Growing Offset Permutation

permutation. The first generation behaves exactly as a normal permutation as depicted in Figure 4. On the right hand side the primary keys of the table can be seen, e.g. employee identifiers. In generation 0 there are four initial elements. In the first generation two elements are deleted, two are changed and two are added. For generation 1, we define three ranges Delete[1, 2], Change[3,4] and New[5,6]. The change part will be important later in Section 4.6 and will be excluded for now. All IDs selected by the permutation from left to right in the range Delete[1,2] are deleted in generation 1. In our example these are the IDs {4,1}.

The permutation for generation 2 maps to generation 1 and not directly to the ID range. It also adds an offset, as the values [1,2] in generation 1 are no longer available since they were deleted. In Figure 5 generation 2 maps $1 \rightarrow 1$ and $2 \rightarrow 4$ but two elements where removed at generation 1. To address this fact an offset is added to the mapping, the correct elements in generation 1 are: $1 \rightarrow (1+2) = 3$ and $2 \rightarrow (4+2) = 6$. To find the real values for 3 and 6 we have to use the first generation's permutation to the real ID. As result, ID 2 and 5 are determined to be deleted in generation 2. The mapping chain for generation 2 is: $1 \rightarrow 1+2 = 3 \rightarrow 2$ and $2 \rightarrow 4+2 = 6 \rightarrow 5$. For generation 3 it is $1 \rightarrow 4+2 = 6 \rightarrow 8$ and $2 \rightarrow 2+2 = 4 \rightarrow 2+2 = 4 \rightarrow 3$. Notice that in the chain for 1 in generation 3 we skip generation 1, as 1 in generation 3 maps to 6 in generation 2 and 6 is new in generation 2. Therefore it cannot have a mapping to generation 1 as this ID has not yet existed in generation 1. This schema guarantees that deleted ID are not picked again (e.g. for updates) and it makes it possible to grow permutations in each generation by adding an arbitrary amount of new elements.

4.6 Update Black Box

In order to encapsulate the evolution of the data over time PDGF 2.0 features an *update black box*. The update black box implements the previously described time-sliced based data evolution based on the Growing Offset Permutation. Each abstract time slice is identified by an ID, in the following called *updateID*. Consider the generation of data for a table containing *cutomers* records, e.g. *c_ID*, *name*, *address*. The attribute ID uniquely identifies a customer. There can be three different update *actions*: *new, change, delete*. New customers are inserted into this table with a given likelihood in a *new* action. Customers are updated in a *change* action that occurs with a certain likelihood. For every *change* action one or more values in the customers tuple change, e.g the address. Obviously, not all values change always at the same time. For example the ID will never change and the name is very unlikely to change, while a fictitious field *last change* would change every time. The end of the lifecycle of a customer record is marked by *delete* action. When the record is deleted no more changes can happen to a customer record and the ID will not be reused again. The update black box models and manages the lifecycle of all IDs in one table. For an ID the cycle is: $new \rightarrow new_{0...n} \rightarrow delete$. As mentioned above, the time granularity is defined by the length of one time slice that is identified by a *update ID*. The time slice is the abstract minimum length of time and can have an arbitrary fixed-sized duration. In every time slice a given number of records - identified by their IDs - are affected by one of the three update actions. For each actions the correct number of records will be added, changed, or deleted in each cycle depending on the actions likelihood. The update black box can calculate the state of every ID for every time slice. It can therefore decide if a certain ID exists, is inserted, deleted or updated in this time slice. Furthermore, the update black box can calculate in which time slice a record was inserted, deleted or updated. This is necessary to retrieve the current state of a record.

The described update black box is integrated in PDGF 2.0 but can be exchanged/extended through plugins. Key feature of its implementation is the growing offset permutation presented before.

4.7 Generation Workflow

In this section we will give a complete overview of the data generation procedure. In an initial load phase PDGF parses the XML file, builds its internal representation, validates and initializes all modules. In the next step the scheduler selects the next table to generate. The scheduler is responsible to split the workload among the threads, e.g. in a round robin fashion. To allow for different scheduling schemes, PDGF 2.0 features a meta scheduler that enables a specific scheduler per table. The threads receive work units from the scheduler. These work units are the description of the next generation task of a thread. In the most basic example this is a table ID, the first and the last row of the data partition that the thread has to generate. Based on this information the worker constructs the key for the next field, uses the seeding strategy to obtain the seed, and seeds the generator's RNG. Then the thread requests the next value from the generator. The generator usually uses the provided RNG to calculate the value. The thread requests all values within a row from the specified generators and passes them to the responsible output system. The output system formats the row and writes it in a file. This is done for all rows in the partition. After that the thread requests the next work unit until all data is generated.

4.8 Some Performance Considerations

As mentioned in Section 3.1 scalability and efficiency are important design goals for PDGF. Hence, during the implementation of PDGF we constantly watched out for pitfalls that might cause bad performance. For example, object allocation and deallocation can be a serious bottleneck, es-

pecially in systems automatic garbage collection. We use reusable data transfer objects for communication between components, thereby, minimizing allocation and deallocation overhead.

One problem in parallel data generators is the generation of ordered output, as the arrival time and thus the order of generated data for individual threads is not always deterministic. Sorting or synchronizing the write operations of generated data can easily become a serious bottleneck. One approach is to block all threads that generated data that arrives too "'early"' until the right element arrives. This is very inefficient. In some cases caching can alleviate the inefficiency, because it allows fast threads to proceed and sort within the cache. PDGF features an ordering cache based on a lock free concurrent SkipListMap data structure. One is used as a value store for the generated data. A second SkipListMap is used to build a custom PriorityLock. The SkipListMap in the PriorityLock is used as a ordered wait list to put threads into sleep state when they are ahead of the current sliding window. A housekeeper thread drains the cache and notifies the thread at the head of the wait list each time an element is removed from the sorted value cache. Using this caching and locking schema we can a assure a high throughput.

5. PERFORMANCE EVALUATION

High data generation speed is of paramount importance for PDGF 2.0. The data sets that are required for TPC benchmarks can reach 100 terabytes. They are typically generated on multi-core systems and in some cases on clusters of multi-core systems. Hence, PDGF 2.0 is required to scale both up and out with various systems.

High data generation speed is particularly important if data sets are generated right before benchmark execution. This helps in implementing an ad-hoc benchmark, because only the data set characteristics (min, max values, distributions etc.) are known a-prior, but not the data itself.

We evaluate PDGF 2.0 on a 24 core enterprise level SMP server. This server has four X5670 Intel Xeon CPUs with six cores and twelve megabytes cache each. They are clocked at 2.93 GHz. The server has a total of 140 gigabytes main memory. Because we do not have access to a sufficiently fast storage array (24 cores would require about 600 MB/s write speed), we write all data to `\dev\null`.

We generate data for an ETL scenario, similar to the benchmark proposed by Wyatt et. al [25], which is based on the data model of TPC-E. Specifically, we generate a Trading table consisting of six columns: i) LastTradeDate, the date of last completed trading day, ii) SecuritySymbol, symbol of the security, iii) ClosingPrice, closing price of the security on LastTradeDate, iv) HighPrice, highest price for the security on LastTradeDate, v) LowPrice, lowest price for the security on LastTradeDate and vi) Volume, trading volume for the security on LastTradeDate. It models daily trading information of securities on exchanges for multiple years. The table is sorted by its first column, LastTradeDate. SecuritySymbol is a reference (foreign key) to a table holding securities. However, only references to certain securities qualify. The securities table does not need to be generated in order to choose qualifying security references. ClosingPrice and HighPrice are random numbers with a specific distribution in different ranges and LowPrice and Volume are intra-column references to ClosingPrice.

For the above table we generate the initial data and 3 update sets, each representing data for one trading day. The update sets contain two additional fields, because they represent data from a change data capture (CDC) system. The first field denotes whether the data is an insert, update or delete. The second represents a database sequence number, basically an incrementally increasing ID. The historical data set includes one record per active security per day between a start date and end date. Each incremental data sets contains one record per active security. The number of active securities depend on a scale factor. The abstract schema can be seen in the table below:

Field	Comment
CDC ID	1,2,3,... (update only)
CDC Flag	i, u, or d (update only)
Date	Sort order
Reference	To other table
Number	Real with predefined distribution
Number	Integer with predefined distribution
Reference	Intra-table reference * random number
Reference	Intra-table reference * random number

We conduct two experiments. The first tests how well PDGF scales with the number of processor cores on a given system. This is tested by generating the data sets for a ficed scale factor of 1.000.000, which results in a total data set of 18 gigabytes. Starting with a single thread we incrementally double the number of threads to a maximum of 32. Obviously, PDGF can only use as many cores as the given number of threads. Therefore, PDGF is not able to fully utilize the system with less than 24 threads. Figure 6 shows the generation time (solid line) and the throughput (dotted line) for this test. The generation speed scales nicely with the number of threads up to 8 threads. Starting with 16 threads scalability flattens out. The maximum throughput of nearly 180 MB/s is achieved with 16 threads. At 32 threads, which is 8 threads more than the number of physical cores, the throughput starts decreasing. The sublinear speedup is due to the synchronization overhead as well as the competition of generation threads with controlling threads. The update black box in PDGF 2.0 follows a producer-consumer pattern, which contains inherent synchronization points and there are several other threads that contain synchronization points. If these threads have to compete with the generation threads the overall throughput can decrease.

To get so some more insights on the synchronization overhead, we run the same test on a SPARC T3-4 server. This system has 4 T3 CPUs with 16 cores, each clocked at 1.65 GHz with a cache size of 6 MB. Each core has 8 threads resulting in a total of 512 threads, i.e. virtual processors. The system had 512 gigabytes of RAM. On this system we chose a scale factor of 100.000 resulting in 1.8 gigabytes of data. The result of this test are depicted in Figure 7. On the first impression the results are surprising. The data generation scales well up to 32 threads and then slows down dramatically. However, these results verify our previous findings. As soon as the generation threads compete with the control threads the generation speed decreases. An interesting side effect is the better performance of 512 threads. We assume that the thread scheduling works more effective if all hardware threads are utilized.

In our second experiment we generated three different data set sizes with 32 threads. We used scale factors 1.000.000,

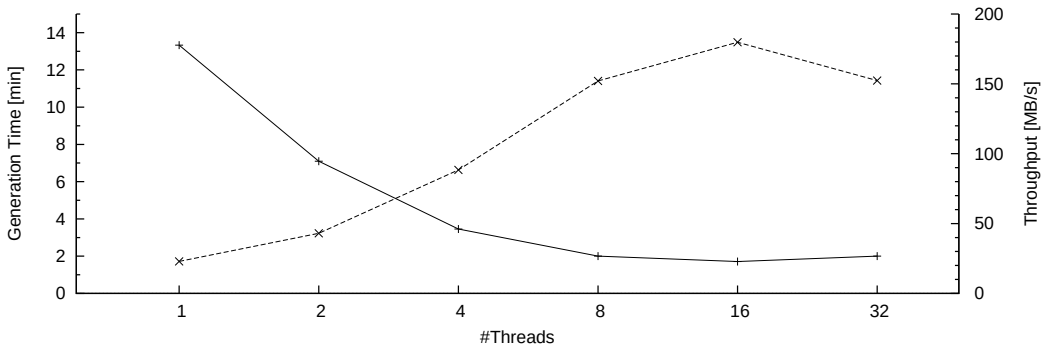

Figure 6: Generating 18 Gigabytes with Different Numbers of Threads

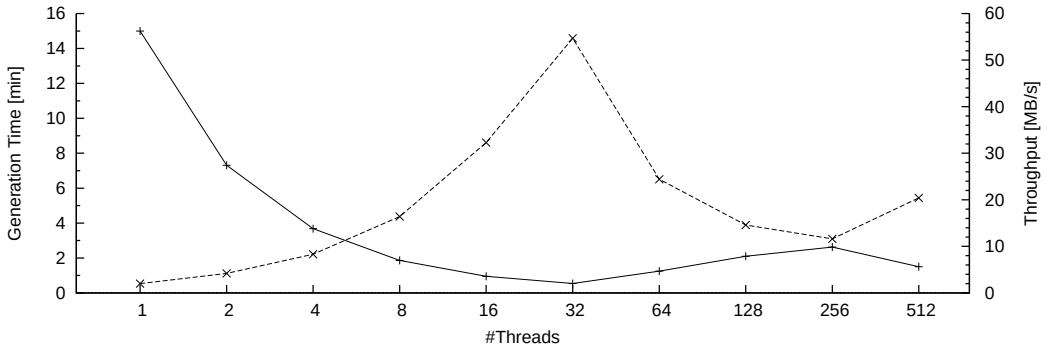

Figure 7: Generating 1.8 Gigabytes with Different Numbers of Threads

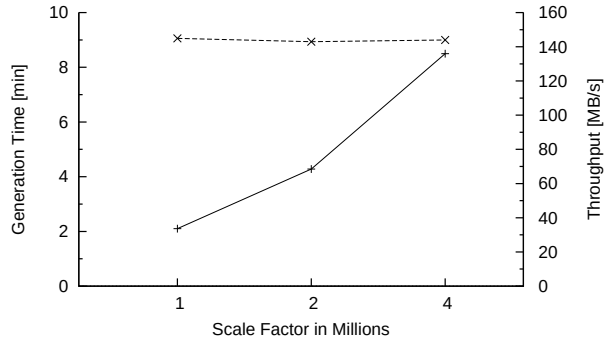

Figure 8: Generating 18-72 Gigabytes with 32 Threads

2.000.000, and 4.000.000; resulting in 18, 36, and 72 gigabytes of data. In Figure 8 the generation time (solid line) and the throughput (dotted line) can be seen for each run. The throughput is fairly constant at 144 MB/s, consequently the generation time is directly dependent to the scale factor. This indicates that it is possible to generate arbitrary large data sets without decreasing throughput. With 144 MB/s it would take less than 2 hours to generate data sets of one terabyte.

In general, these results present the good scalability of our system. For large SMP systems like our test system the synchronization overhead is reasonable, but it decreases the overall speedup at high thread counts. However, this is an example of a fairly complex relation. Most tables are less complicated to generate and will therefore have much better throughput and scalability.

6. CONCLUSIONS

In this paper we have presented our approach to data generation that makes it possible to generate consistent updates and change data captures. Our approach is based on the parallel data generation framework, PDGF, Version 1.0. The new version is PDGF 2.0. Basis of our method is the exploitation of determinism in random number generation. To generate unique references to we use invertible permutations. To generate consistent updates we have introduced the concept of a growing offset permutation which enables us to keep track of the changes in the database when we generate update, inserts and deletes. Our experiments show that the generation scales perfectly with the problem size and that we can achieve good speedups on large enterprise class server systems.

For future work we will further extend this concept to generate verifiable query workloads. For this we will generate queries using our data generator as references to the base tables. This way we will be able to pre-compute the query results and therefore also test the correctness of the system under test. For this we will combine the deterministic data generation with our workload generation concept [20].

7. ACKNOWLEDGEMENTS

The authors would like to thank Manuel Danisch for his help with the implementation of PDGF 2.0. Furthermore,

we thank the anonymous reviewers for their valuable input that helped to improve the quality of the paper.

8. REFERENCES

[1] C. Bennett, R. Grossman, and J. Seidman. MalStone: A Benchmark for Data Intensive Computing. Technical report, Open Cloud Consortium, 2009.

[2] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking Database Systems: A Systematic Approach. In *VLDB '83: Proceedings of the 9th International Conference on Very Large Data Bases*, pages 8–19, San Francisco, CA, USA, November 1983. ACM, Morgan Kaufmann Publishers Inc.

[3] N. Bruno and S. Chaudhuri. Flexible Database Generators. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Databases*, pages 1097–1107. VLDB Endowment, 2005.

[4] Continuent. Bristlecone. https://bristlecone.svn.sourceforge.net/svnroot/-bristlecone/trunk/bristlecone/.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10: Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, New York, NY, USA, 2010. ACM.

[6] DTM Database Tools. Dtm data generator. http://www.sqledit.com/dg/.

[7] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 243–252, New York, NY, USA, 1994. ACM.

[8] GSApps. Gs data generator. http://www.gsapps.com/products/datagenerator/.

[9] J. E. Hoag and C. W. Thompson. A Parallel General-Purpose Synthetic Data Generator. *SIGMOD Record*, 36(1):19–24, 2007.

[10] K. Houkjær, K. Torp, and R. Wind. Simple and Realistic Data Generation. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246. VLDB Endowment, 2006.

[11] K. Huppler. The Art of Building a Good Benchmark. In *TPCTC '09: First TPC Technology Conference on Performance Evaluation and Benchmarking*, pages 18–30, 2009.

[12] P. J. Lin, B. Samadi, A. Cipolone, D. R. Jeske, S. Cox, C. Rendón, D. Holt, and R. Xiao. Development of a Synthetic Data Set Generator for Building and Testing Information Discovery Systems. In *ITNG '06: Proceedings of the Third International Conference on Information Technology: New Generations*, pages 707–712, Washington, DC, USA, 2006. IEEE Computer Society.

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

[14] R. O. Nambiar and M. Poess. The Making of TPC-DS. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1049–1058, 2006.

[15] P. E. O'Neil. The Set Query Benchmark. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann Publishers, 1993.

[16] F. Panneton and P. L'ecuyer. On the Xorshift Random Number Generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, 2005.

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

[18] M. Poess, T. Rabl, M. Frank, and M. Danisch. A PDGF Implementation for TPC-H. In *TPCTC '11: Third TPC Technology Conference on Performance Evaluation and Benchmarking*, 2011.

[19] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A Data Generator for Cloud-Scale Benchmarking. In *TPCTC '10:Proceedings of the Second TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 41–56, 2010.

[20] T. Rabl, A. Lang, T. Hackl, B. Sick, and H. Kosch. Generating Shifting Workloads to Benchmark Adaptability in Relational Database Systems. In R. O. Nambiar and M. Poess, editors, *TPCTC '09: First TPC Technology Conference on Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2009.

[21] T. Rabl and M. Poess. Parallel Data Generation for Performance Analysis of Large, Complex RDBMS. In *DBTest '11: Proceedings of the 4th International Workshop on Testing Database Systems*, page 5, 2011.

[22] Red Gate. Sql data generator 2.0. http://www.red-gate.com/products/sql-development/sql-data-generator/.

[23] J. M. Stephens and M. Poess. MUDD: a multi-dimensional data generator. In *WOSP '04: Proceedings of the 4th International Workshop on Software and Performance*, pages 104–109, New York, NY, USA, 2004. ACM.

[24] The Transaction Performance Processing Council. Dbgen. http://www.tpc.org/tpch/.

[25] L. Wyatt, B. Caufield, and D. Pol. Principles for an ETL Benchmark. In *TPC TC '09: First TPC Technology Conference on Performance Evaluation and Benchmarking*, pages 183–198, 2009.