A Data Generator for Cloud-Scale Benchmarking

Tilmann Rabl, Michael Frank, Hatem Mousselly Sergieh, and Harald Kosch

Chair of Distributed Information Systems
University of Passau,
Germany
{rabl,frank,moussell,kosch}@fim.uni-passau.de
http://www.dimis.fim.uni-passau.de

Abstract. In many fields of research and business data sizes are breaking the petabyte barrier. This imposes new problems and research possibilities for the database community. Usually, data of this size is stored in large clusters or clouds. Although clouds have become very popular in recent years, there is only little work on benchmarking cloud applications. In this paper we present a data generator for cloud sized applications. Its architecture makes the data generator easy to extend and to configure. A key feature is the high degree of parallelism that allows linear scaling for arbitrary numbers of nodes. We show how distributions, relationships and dependencies in data can be computed in parallel with linear speed up.

1 Introduction

Cloud computing has become an active field of research in recent years. The continuous growth of data sizes, which is already beyond petabyte scale for many applications, poses new challenges for the research community. Processing large data sets demands a higher degree of automation and adaptability than smaller data sets. For clusters of thousand and more nodes hardware failures happen on a regular basis. Therefore, tolerance of node failures is mandatory. In [19] we sketched a benchmark for measuring adaptability, in this paper we present a data generator that is cloud aware, as it is designed with the top goals of cloud computing, namely scalability and decoupling, i.e. avoidance of any interaction of nodes [3].

Traditional benchmarks are not sufficient for cloud computing, since they fall short on testing cloud specific challenges [2]. Currently, there are only a few benchmarks available specifically for cloud computing. The first one was probably the TeraSort Benchmark[1]. Others followed, such as MalStone and CloudStone. These benchmarks are dedicated to a single common task in cloud computing. While this kind of benchmarks is essential for scientific research and evaluation, it fails to give a holistic view of the system under test. We think

[1] The current version can be found at http://www.sortbenchmark.org/

that there is a need for a benchmark suite that covers various aspects of cloud computing. The database community has traditionally an affection for simple benchmarks [11, 22]. Although reduction of complexity is a basic principle of computer science and unnecessary complexity should be avoided by all means, there seems to be a trend to simplistic evaluations. In order to get meaningful results benchmarks should have diverse and relevant workloads and data [5]. Often it is best to use real life data. For example, in scientific database research the Sloan Digital Sky Survey is frequently referenced [24]. Yet for many applications such as social websites there is no data publicly available. And even though storage prices are dropping rapidly, they are still considerably high for petabyte scale systems. A current 2 TB hard drive costs about USD 100, resulting in a GB price of about USD 0.05. So the price for 1 PB of raw disk space is about USD 50000. Therefore, it is not sensible to store petabytes of data only for testing purposes. Besides storage, the network necessary to move petabytes of data in a timely manner is costly. Hence, the data should be created where it is needed. For a cluster of nodes this means that each node generates the data it will process later, e.g. load into a data base. In order to generate realistic data, references have to be considered, which usually requires reading already generated data. Examples for references are foreign keys. For clusters of computers this results in a fair amount of communication between nodes.

For example consider a table representing a many-to-many relationship between two tables. When generating corresponding keys one needs information about the keys in the two tables participating in the many-to-many relationship. On a single node system it is usually much faster to read in the two tables to create the relationship. But if the data for the two tables is scattered across multiple nodes, it is more efficient to regenerate it. This way the relationship can be created completely independently of the base tables. By using distributed parallel random number generators, such as the leap frog method [10, Ch.10], even the generation of single tables can be parallelized. Since the generation is deterministic, also references can still be computed independently.

In this paper we present an ubiquitous parallel data generation framework (PDGF) that is suitable for cloud scale data generation. It is highly parallel and very adaptable. The generator uses XML configuration files for data description and distribution. This simplifies the generation of different distributions of specified data sets. The implementation is focused on performance and extensibility. Therefore, generators for new domains can be easily derived from PDGF. The current set of generators is built to be nearly write-only, so they generate values only according to random numbers and relatively small dictionaries, but without rereading generated data. This is done to reduce I/O and network communication to the absolute minimum.

Consider the simplified example in Figure 1, taken from the eLearning management system presented in [19]. There are three tables, *user*, *seminar*, and *seminar_user*. The generation of tables *user* and *seminar* are straightforward. For *seminar_user* it has to be considered, that only *user_id*s and *seminar_id*s are generated, that actually exist in *user* and *seminar*. This is only easy if both

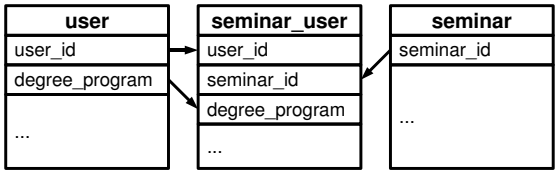

Fig. 1. Fragment of the schema of an eLearning management system.

attributes have continuous values, otherwise it has to be assured that the referenced tuples exist. A second challenge is that *degree_program* is replicated in *seminar_user*, so the combination of *user_id* and *degree_program* have to exist in *user*. Finally the values of tables like *seminar_user* have typically non uniform distributions.

The common solution to generate the table *seminar_user* is to first generate the two tables that are referenced and then use a look up or scan to generate the distribution. If this is done in parallel, either the referenced data has to be replicated, or the data generating process has to communicate with other nodes. This is feasible for smaller clusters, but for cloud scale configurations the communication part will be the bottleneck. Therefore, we propose a fully computational approach. Basically, our data generation is a set of functions that map a virtual row id to a tuple's attribute. Using this approach, we can easily recompute every value. So for the example above we would define a function for each attribute in the original tables. To generate uncorrelated data, the first computational step is usually either a permutation or a pseudo random number generation. For the example above this would only be needed for the *degree_program*. The value could either be chosen from a dictionary or be generated. To generate entries of *seminar_user*, two pseudo random numbers in the range of $[1, |user|]$ and $[1, |seminar|]$ are computed, with according distribution properties and then the function to generate *degree_program* is used, resulting in a valid tuple for *seminar_user*. This can be computed completely independently of the generation of *user* and *seminar*. Since parallel pseudo random number generators are used, *seminar_user* can be generated on any reasonable number of computing nodes in parallel.

This flexibility opens a broad field of application. Besides traditional relational, row oriented data, our system can easily generate data in other storage models, such as the Decomposed Storage Model [9], column wise as in MonetDB [6] or C-Store [23] or even mixed models [20].

An often discussed problem is that experiments are run on too small data sets [15]. This is not because of the negligence of researchers, but because large sized experiments are very time consuming and many researchers have no access to unlimited sized storage clusters. Hence, in many situations it seems sensible to use only a fraction of the data set and simulate a fraction of the system. Obviously, there are many research fields where such a methodology leads to realistic results. An example would be sort experiments. Our data generator is capable of generating statistically sound extracts of data sets as well as it can

generate large data sets in uncorrelated fractions. So large scale test can either be scaled down or processed sequentially.

The rest of the paper is organized as follows, Section 2 gives an overview of previous work on data generation for database benchmarking. After that, Section 3 describes how the design goals of the data generator are met. Section 4 explains the data generation approach. The architecture of the data generator is described in Section 5. We evaluate the performance of our data generator in Section 6 and conclude with future work in Section 7.

2 Related Work

There has been quite some research on data generation for performance benchmarking purposes. An important milestone was the paper by Gray et al. [12], 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.

According to their reference generation procedure, data generators can be roughly divided into three categories: no reference generation, scanning references, and computing references. No reference generation means that no relationships between tables are explicitly considered. So references are either only simple or based on mathematical probabilities. In this scheme it is for example not possible to generate foreign keys on a non-continuous unique key. Examples are data sets that only consists of single tables or data sets (e.g. SetQuery [17], TeraSort, MalGen [1], YCSB [8]) or unrelated tables (e.g. Wisconsin database [4], Bristlecone[2]).

Scanned references are generated reading the referenced tuple, this is either done simultaneously to the generation of the referenced tuple or by scanning the referenced table. This approach is very flexible, since it allows a broad range of dependencies between tables. However, the generation of dependent tables always requires the scanning or calculation of the referenced table. When the referenced table is read, additional I/Os are generated, which in many applications will limit the maximum data generation speed. Generating tables simultaneously does not constitute a problem. However, it requires generating all referenced tables. This is very inefficient, if the referenced tables are very large and don't need to be generated, e.g. for an materialized view with aggregation. Most systems that generate references use scanned references. An example is dbgen[3], the data generator provided by the TPC for the TPC-H benchmark[18]. Another approach was presented by Bruno and Chaudhuri [7], it largely relies on scanning a given database to generate various distributions and interdependencies. In [14] Houkjær et al. describe a graph based generation tool, that models dependencies in a graph and uses a depth-first traversal to generate dependent tables. A

[2] Available at `http://www.continuent.com/community/lab-projects/bristlecone`
[3] dbgen can be downloaded from `http://www.tpc.org/tpch/`

similar approach was presented by Lin et al. [16]. Two further tools that offer quite similar capabilities are MUDD [21] and PSDG [13]. Both feature description languages for the definition of the data layout and advanced distributions. Furthermore, both tools allow parallel generation. However, as described above the independent generation of dependent data sets is not possible.

A computed reference is recalculated using the fact, that the referenced data is deterministically generated. This results in a very flexible approach that also makes it possible to generate data with cyclic dependencies. The downside is the computational cost for regenerating the keys. However, as our test shows (see Section 6) current hardware is most often limited by I/O speed. To the best of our knowledge our tool is the only one that relies on this technique for parallel data generation.

3 Design Goals

PDGF's architecture was designed with the following goals: platform independence, extensibility, configurability, scalability and high performance. The following sections explain how each goal is met.

3.1 Platform Independence

To achieve a high degree of platform independence, PDGF was written in Java. It has been tested under Windows and different Linux distributions. Using Java has no degrading effect on the performance of the generator. In fact, startup and initialization times can be neglected compared to the time taken to generate terabytes of data for the cloud. This characteristic gives the just-in-time compilation technology enough time to compile and optimize the code, so that a similar generation speed as with pre-compiled binaries can be achieved.

3.2 Extensibility

PDGF is a framework that follows a black box plug-in approach. PDGF components can be extended and exchanged easily. Custom plug-ins can be implemented without consideration of programming aspects like parallelism or internal details. Furthermore, some minor built-in components are also exchangeable via the plug-in approach. These components include the file caching strategies and the scheduling strategy which are responsible for work splitting among threads and nodes. To make the system aware of new plug-ins it is sufficient to place them in the classpath and to reference them in database scheme description file.

3.3 Configurability

PDGF can be configured by two XML-based configuration files. One file configures the runtime environment, while the other configures the data schema and generation routines.

Runtime Configuration File The runtime configuration file determines which part of the data is generated on a node and how many threads are used. This is used for splitting the work between participating nodes. The file is optional, since these settings can also be specified via command line or within the built-in mini shell. Listing 1 shows a sample runtime configuration file. This example is for Node 5 out of 10 nodes. Two worker threads are used.

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeConfig>
 <nodeNumber>5</nodeNumber>
 <nodeCount>10</nodeCount>
 <workers>2</workers>
</nodeConfig>
```

Listing 1. "Runtime configuration file"

Data Schema Configuration File The data schema configuration file is used to specify how the data is generated. It follows a hierarchical structure as illustrated below. As mentioned above, all major components (e.g. the random number generator, data generators, etc.) can be exchanged by plug-ins. To use a plug-in, its qualified class name has to be specified in the name attribute of the corresponding element (see listing 2).

```
<project name="simpleE-learning">
 <scaleFactor>1</scaleFactor>
 <seed>1234567890</seed>
 <rng name="DefaultRandomGenerator" />
 <output name="CSVRowOutput">
  <outputDir>/tmp</outputDir>
 </output>
 <tables>
  <table name="user">
   <size>13480</size>
   <fields>
    <field name="user_id">
     <type>java.sql.Types.INTEGER</type>
     <primary>true</primary> <unique>true</unique>
     <generator name="IdGenerator" />
    </field>
    <field name="degree_program">
     <type>java.sql.Types.VARCHAR</type>
     <size>20</size>
     <generator name="DictList">
      <file>dicts/degree.dict</file>
     </generator>
    </field>
   </fields>
  </table>
  <table name="seminar">  [..]  </table>
```

```
  <table name="user_seminare">
   <size>201754</size>
   <fields>
    <field name="user_id">
     <type>java.sql.Types.INTEGER</type>
     <reference>
      <referencedField>user_id</referencedField>
      <referencedTable>user</referencedTable>
     </reference>
     <generator name="DefaultReferenceGenerator">
      <distribution name="LogNormal">
       <mu>7.60021</mu> <sigma>1.40058</sigma>
      </distribution>
     </generator>
    </field>
    <field name="degree_program"> [..] </field>
    <field name="seminar_id">     [..] </field>
   </fields>
  </table>
 </tables>
</project>
```

Listing 2. "Config file example for userSeminar references."

3.4 Scalability

To work in cloud scale environments a data generator must have a deterministic output regardless the number of participating nodes. Moreover, it is necessary to achieve a nearly linear performance scaling with an increasing number of participating nodes.

To fulfill these requirements, we followed an approach that avoids the usage of a master node or inter-node and inter-process communication. Every node is able to act independently and process its share of workload without considering the data generated by other nodes. To achieve this kind of independence, a combination of multiple instances of random number generators with inexpensive skip-ahead functionality was applied. Allowing each node to determine its workload is done as follows: Each node starts a process and initializes it with the total number of participating nodes in addition to the node number. The workload is divided into work units. Each work unit contains the current table to be processed, the start row number and the end row number. For example, assuming that we have a table T1 with 1000 rows and 10 nodes (node1 to node10), then the workload is divided as follows:

node1 work unit 1..100
node2 work unit 101..200
..
node10 work unit 901..1000

The same approach is applied for every table of the database schema. Of course PDGF also utilizes modern multi-core CPUs by spawning a worker thread per (virtual) core. The number of worker threads is automatically determined or can be set manually in the configuration. After the workload has been distributed on the nodes, the work unit of a table is further divided and distributed on the worker threads of each node using the same approach. However, the start value of the work unit of the node should be added to the start and the end of each worker workload. For instance, assume that Node 2 has four worker threads. The work unit will be split among the worker threads.

node2 work unit 101..200
    worker1 work unit 101..125
    worker2 work unit 126..150
    worker3 work unit 151..175
    worker4 work unit 176..200

3.5 High Performance

PDGF achieves high performance by applying a set of strategies that allow efficient computation by minimizing the amount of I/O operations. To avoid synchronization overhead, custom per thread buffering and caching is used. Furthermore, the overhead caused by object creation was reduced by applying eager object initialization and the strict usage of shared reusable data transfer objects on a per thread basis.

Recalculating the value for a field on demand is usually less expensive than reading the data back into memory, as can be seen in the evaluation below. The static scheduler also generates no network traffic during the data generation. Avoiding read operations on the generated data limits I/O reads to the initialization stage. Only data generators that require large source files (i.e. dictionaries for text generation) produce I/O read operations. Therefore, an abstraction layer between the framework and the files was added to provide direct line access through an interface. This layer can easily be extended to use new file caching strategies. In the current implementation all files, independent of their size, are completely cached in the memory during the startup phase. With current hardware this is no problem for files up to hundreds of megabytes. There are many other aspects that directly or indirectly aid the performance, e.g. the efficient seeding strategy or the fast random number generator.

4 Data Generation Approach

To generate field values independently and to avoid costly disk reads, a random number generator (RNG) and a corresponding seed are assigned to each field. The used RNGs are organized hierarchically so that a deterministic seed for each field can be calculated, see Figure 2. To calculate a value for a field in a

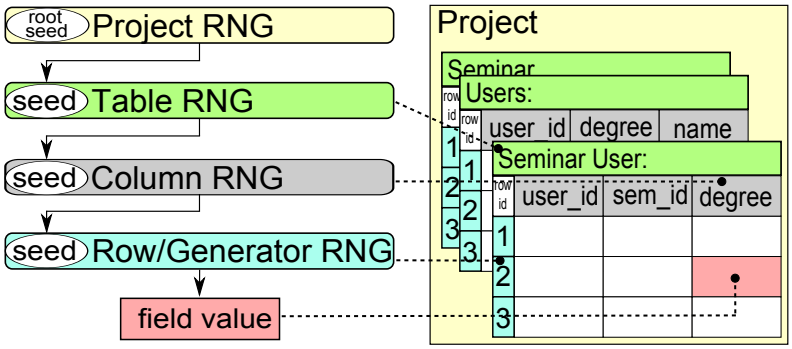

Fig. 2. Seeding strategy of PDGF

specific row the field's RNG only has to be re-seeded with the corresponding deterministic seed.

The RNG of the project element is considered as the root RNG and its seed is considered as the root seed. Each seed is used to derive the value of the seeds of the RNGs at the next lower level of the hierarchy. For example, the project seed is used to calculate the seed of the RNG of the tables and that of a table is used to generate seeds for columns and so on. Since there is only one project seed, all other seeds can be derived from that seed. As table and field (column) count are static, their seeds are cached after the first generation. Usually PDGF does not need to run through the complete seeding hierarchy to determine the seed for the generator RNG. It is sufficient to re-seed the field RNG with its pre-calculated seed, skip forward to the row needed and get the next value. For the default RNG used in PDGF this is a very inexpensive operation. After seeding, the RNG is passed to the corresponding field generator to generate the actual value.

5 Architecture

Figure 3 shows the architecture of the PDGF data generator. It consists of 5 basic components:

- The controller
- the view
- the generators, which contain
  - field generators
  - distribution functions
  - the random number generator
- The output module for generated data
- The scheduler

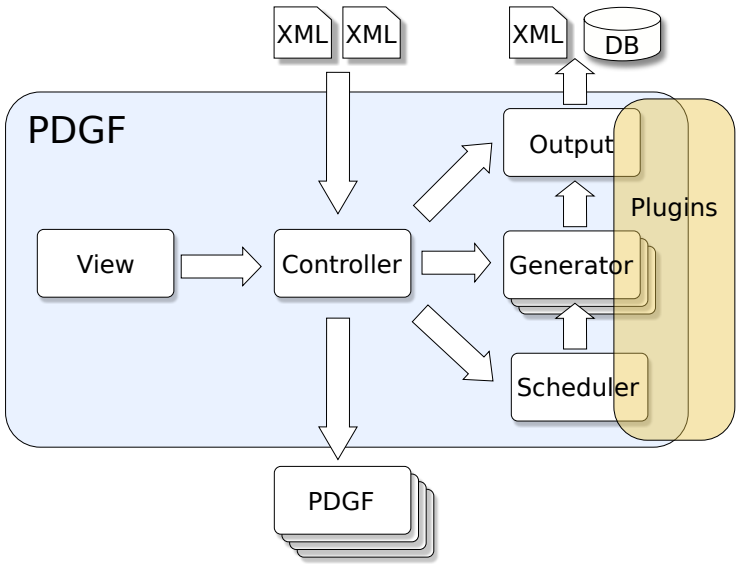

Fig. 3. Architecture of PDGF

Controller/View The controller takes user input such as the configuration files from the command line or a built in interactive mini-shell. This input is used to configure an instance of PDGF and to control the data generation process. By default PDGF is a stand alone command line tool but other views can be easily attached to the controller e.g. a GUI. The controller allows the use of PDGF as a library. Distributed startup is currently realized by an external shell script, to allow the use with queuing systems. As PDGF can be used as a library, it is possible to use more complex startup algorithms.

Field Generators The field generators are the most important part as they determine how the values for a field are generated. PDGF comes with a set of generators for general purpose: data, default reference, dictionary, double values, ids, int values, pseudo text grammar and static values. Since some data sets require a special structure, e.g. the TPC-H benchmark data set, PDGF provides a simple interface enabling easy implementation of generator plug-ins without problems with parallelization or concurrency.

Distribution Functions The distribution functions allow generators to easily adapt and exchange how the values will be distributed. The distribution function uses the uniform random values from the RNG provided to each generator to calculate the non-uniformly distributed values. As for the field generators PDGF comes with some basic distribution functions: beta, binomial, exponential, log-normal, normal, Poisson, and Student's-t.

Random Number Generator A parallel random number generator is the key to make the seeding algorithm efficient and fast. The used RNG generates random

numbers by hashing successive counter values where the seed is the initial value for the counter. This approach makes skipping ahead very inexpensive. The default RNG can also be exchanged and in addition it is even possible to specify a custom RNG per Generator. As for all other plug-ins, there is an interface that allows the usage of other RNGs.

Output Module The output module determines how to save the generated data. An output module receives the values of an entire row for a table along with some meta information. By default the generated data is written to a comma separated value file, one per table and instance. Another possibility is to convert the generated values into SQL insert statements. These can either be written to a file or sent directly to a DBMS. In contrast to the other plug-in types an output plug-in is a serialization point of PDGF as all workers write concurrently to it.

Scheduler The scheduler is the most complex plug-in. It influences the framework behavior to a large degree. The scheduler is responsible for dividing the work among physical nodes and the worker threads on the nodes. PDGF's default strategy is to statically divide the work between nodes and workers in junks of equal size. This is efficient if the data is generated in a homogeneous cluster or similar environment. In a heterogeneous environment the static approach leads to varying elapsed times among the participating nodes.

6 Performance Evaluation

We use the TPC-H and the SetQuery databases to evaluate our data generator [18, 17]. All tests are conducted on a high performance cluster with 16 nodes. Each node has two Intel Xeon QuadCore processors with 2 GHz clock rate, 16 gigabyte RAM and two 74 GB SATA hard disks configured with RAID 0. Additionally, a master node is used, which has the same configuration, but an additional hard disk array with a capacity of 2 TBytes. For both benchmarks two test series are conducted. The first series tests the data generator's scalability in terms of data size on one node. The second series demonstrates the data generator's scalability in terms of the number of nodes with fixed data size. Each test was repeated at least five times. All results are averages of these test runs.

SetQuery The SetQuery data set consists of a single table *BENCH* with 21 columns. 13 of the columns are integers with varying cardinalities from 2 to $1,000,000$ of which 12 are generated randomly. 8 additional columns contain strings, one with 8 characters and the others with 20 characters. The table size is scaled linearly according to a scaling factor SF, where $SF = 1$ results in about 220 MB. First we generated a 100 GB data set on 1 to 16 nodes (i.e. scaling factor 460). As can be seen in Figure 4 the average speed up per node is linear to the number of nodes. One node was able to generate about 105 MB per second.

The super linear speed up for a higher number of nodes results from caching effects, which can be seen more clearly in the second test.

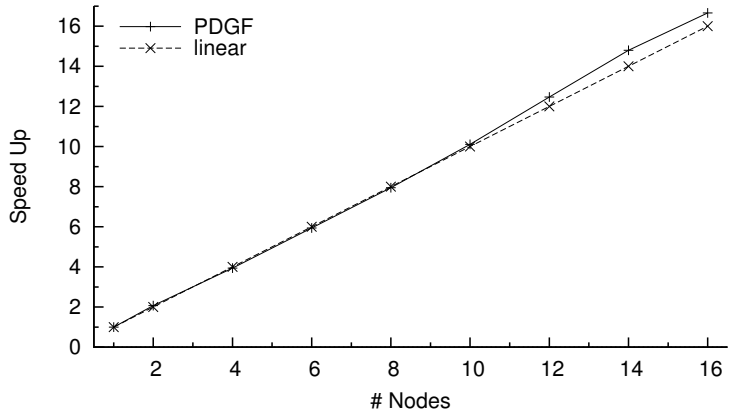

Fig. 4. Scaleup results for 1 to 16 nodes for a 100 GB SetQuery data set

For the second test, different data sizes are generated on a single node. We use scale factor 1 to 460. The resulting elapsed times for data generation are shown in Figure 5. It can be seen that the data generation scales well with the amount of data. However, the generation is not constant. This is due to caching effects and initialization. For smaller data sizes the initialization overhead decreases the overall generation speed. Then at scaling factor 10 (i.e. about 2 GB) there is a peak that results from hard disk and disk array caches. For larger data sizes the hard disks write speed is the bottleneck and limits the generation speed to about 100 MB/s.

TPC-H To test a more complex scenario and compare the generation speed with other data generation tools, we used our data generator to generate TPC-H data. TPC-H defines 8 tables with different sizes and different number of columns. The schema defines foreign key relations and the following data types: integer; char; varchar; decimal and date. The amount of data is scaled linearly with the scale factor SF, where $SF = 1$ will result in 1 GB of data. Again, we tested the data generator's scalability in terms of the amount of data and the number of nodes. Figure 6 shows the data generation elapsed times for scale factor 1, 10 and 100 for a single node. Additionally, we generated the same data sizes with dbgen. Both axes of the figure are in logarithmic scale. To obtain fair results, dbgen was started with 8 processes, thus fully exploiting the 8 core system. Generation times for both tools were CPU bound. Since we had notable variations in the runtime of dbgen, we only report the best of 5 runs for each scaling factor. As can be seen in the figure, our configurable and extensible Java implemented tool can compete with a specialized C implementation.

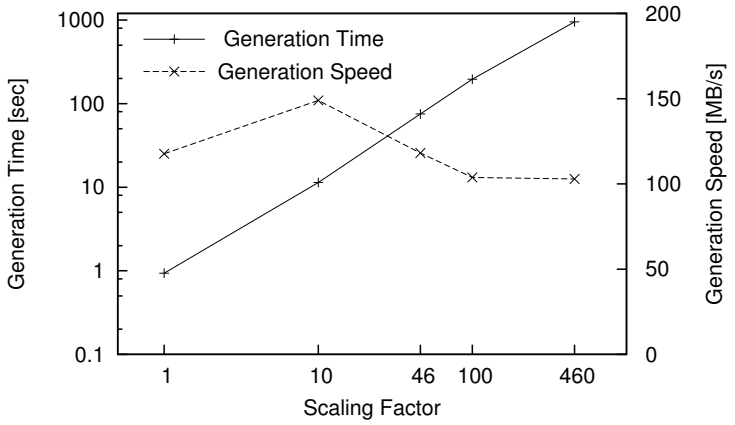

Fig. 5. Generation time and speed for different scaling factors of the SetQuery data set

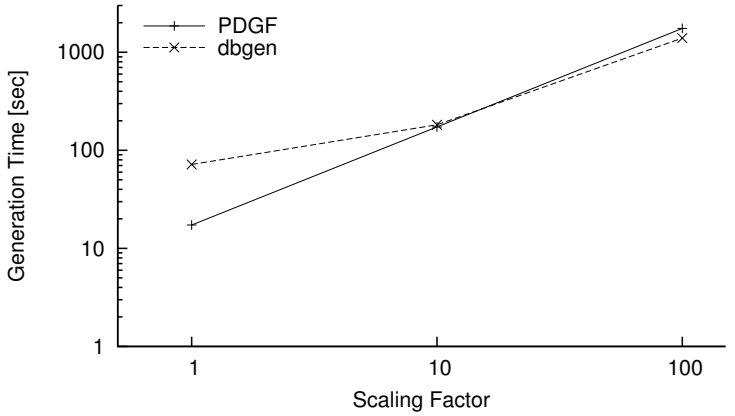

Fig. 6. Comparison of the generation speed of dbgen and PDGF

Figure 7 shows the data generation elapsed times for different cluster sizes. For all cluster sizes the data generation elapsed time scales linearly with the data size. Furthermore, the generation time for certain scale factors decreases linearly with the number of nodes it is generated on. However, for scale factor 1 on 10 and 16 nodes the generation speed is significantly slower than for the other configurations. This is due to the large initialization overhead compared to the short generation time.

E-Learning To measure data generation speed for more complicated distributed values we executed our simple example with the configuration file shown in 2. Even in this example with log normal distributed values and reference calcula-

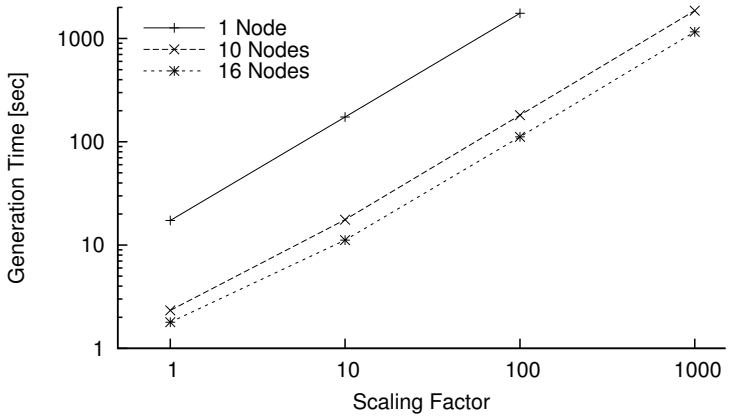

Fig. 7. Generation times of TPC-H data sets on different cluster sizes

tion, the generation speed is only limited by the hard disk speed. The values are therefore similar to the SetQuery results.

7 Conclusion

In this paper we presented a framework for parallel data generation for benchmarking purposes. It uses XML files for the data definition and the configuration file. Like other advanced data generators (e.g. [21, 7, 13, 14]) it features dependencies between relations and advanced distributions. However, it uses a new computational model, which is based on the fact that pseudo random numbers can be recomputed deterministically. Using parallel pseudo random number generators, dependencies in data can be efficiently solved by recomputing referenced data values. Our experiments show, that this model allows our generic, Java implemented data generator to compete with C implemented, specialized data generators.

For future work we are intending to further expand our set of generators and distributions. Furthermore, we will implement a GUI to allow a more convenient configuration. We also want to include other features, as for example schema and distribution retrieval from existing databases. To further increase the performance, we will include new schedulers that reduce wait times for slower nodes, as well as caching strategies to reduce re-computation of repeatedly used values.

To complete our benchmarking suite, we will use the data generator to implement a query generator. For this we will introduce time series generators. This will enable the generation of varying query streams as we presented in [19]. Furthermore, it will enable realistic time related data generation.

8 Acknowledgement

The authors would like to acknowledge Meikel Poess for his helpful comments and feedback on earlier versions of the paper.

References

1. C. Bennett, R. Grossman, and J. Seidman. Malstone: A benchmark for data intensive computing. Technical report, Open Cloud Consortium, 2009.
2. C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow?: Towards a benchmark for the cloud. In *DBTest '09: Proceedings of the Second International Workshop on Testing Database Systems*, pages 1–6, New York, NY, USA, 2009. ACM.
3. K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.
4. 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.
5. S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.
6. P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. In *VLDB '09: Proceedings of the 35th International Conference on Very Large Data Bases*, pages 1648–1653. VLDB Endowment, 2009.
7. N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1097–1107. VLDB Endowment, 2005.
8. 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.
9. G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 268–279, New York, NY, USA, 1985. ACM.
10. I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison Wesley, 1995.
11. J. Gray. Database and transaction processing performance handbook. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition).* Morgan Kaufmann Publishers, 1993.
12. 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.
13. J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator. *SIGMOD Record*, 36(1):19–24, 2007.

14. 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.
15. H. F. Korth, P. A. Bernstein, M. F. Fernández, L. Gruenwald, P. G. Kolaitis, K. S. McKinley, and M. T. Özsu. Paper and proposal reviews: is the process flawed? *SIGMOD Record*, 37(3):36–39, 2008.
16. 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.
17. 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.
18. M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. *SIGMOD Record*, 29(2000):64–71, 2000.
19. 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.
20. R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 430–441. VLDB Endowment, 2002.
21. 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.
22. M. Stonebraker. A new direction for tpc? 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 11–17. Springer, 2009.
23. M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564. VLDB Endowment, 2005.
24. A. S. Szalay, J. Gray, A. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg. The sdss skyserver: Public access to the sloan digital sky server data. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 570–581, New York, NY, USA, 2002. ACM.