Just can't get enough - Synthesizing Big Data

Tilmann Rabl
Middleware Systems Research Group
University of Toronto
Canada
tilmann.rabl@utoronto.ca

Manuel Danisch, Michael Frank, Sebastian Schindler
bankmark UG
Passau, Germany
{first.last}@bankmark.de

Hans-Arno Jacobsen
Middleware Systems Research Group
University of Toronto
Canada
jacobsen@eecg.toronto.edu

ABSTRACT

With the rapidly decreasing prices for storage and storage systems ever larger data sets become economical. While only few years ago only successful transactions would be recorded in sales systems, today every user interaction will be stored for ever deeper analysis and richer user modeling. This has led to the development of big data systems, which offer high scalability and novel forms of analysis. Due to the rapid development and ever increasing variety of the big data landscape, there is a pressing need for tools for testing and benchmarking.

Vendors have little options to showcase the performance of their systems but to use trivial data sets like TeraSort or WordCount. Since customers' real data is typically subject to privacy regulations and rarely can be utilized, simplistic proof-of-concepts have to be used, leaving both, customers and vendors, unclear of the target use-case performance. As a solution, we present an automatic approach to data synthetization from existing data sources. Our system enables a fully automatic generation of large amounts of complex, realistic, synthetic data.

1. INTRODUCTION

Data generation is a tedious part of the daily routine of researchers testing new algorithms, database administrators testing new configurations, and performance engineers testing new optimizations. Typically, the data is generated using scripting solutions, which are written and rewritten for each individual use case. While this is manageable for simple and small data sets, databases are ever increasing in size and complexity. Database testing typically is done on simple scenarios, however, customers demand ever more realistic benchmarks that match their use cases [21]. Modern enterprise systems comprise hundreds to thousands of tables, which frequently have more than fifty columns. System vendors that can show the performance of their systems in a most realistic setup have an advantage over vendors that present generic, simple benchmarks.

To this end, multiple generic data generators have been developed, that enable a fast description of data models and correlations [10, 9, 1, 4]. Although all of these systems feature more efficient and more realistic generation of data none has reached wide adoption in the industry. While building custom data generators for each use case is highly inefficient, generic tools are either too limited in their features or their learning curve is too steep to be accepted by developers. This has led to a dilemma, where it is often infeasible for customers to find the best solution for their use case since they can only give metadata but not real data to vendors and vendors are not able to cover the complex schemas and dependencies in their proof of concept systems.

What is needed is a simple solution, that is able to automatically generate realistic data based only on meta information of the original data. Because of ever increasing data sizes and the prevalent increase of cores, processors, and physical machines in deployed systems, the solution has to be performant and scalable. The data has to be rich in features even in single values, since big data systems do not stop processing at the value level. With these requirements in mind, we have developed DBSynth, a fully automatic data generation solution that can reproduce realistic data sets based on schema information and sampling. Unlike any other generic data generator, it can extract features on the value level and generate new relevant values.

DBSynth is an extension to the Parallel Data Generation Framework (PDGF), a generic data generator suite [17]. PDGF is the basis for the data generator of the new industry standard ETL benchmark TPC-DI, which was released by the Transaction Processing Performance Council (TPC) in January 2014 [16] and was also used to implement the data generator for BigBench, the first end-to-end proposal for a big data analytics [7].

Our main contributions are the following, we demonstrate DBSynth, which is the answer to one of the fundamental challenges in the big data space today - finding the best system for a given use case. DBSynth is the first tool, which can generate complete data models from large complex databases on a variety of systems, sample data sets and automatically create relevant dictionaries and Markov models. Furthermore, we demonstrate PDGF, a highly advanced generic data generator, which can use models created by DBSynth to create big, realistic, synthetic data. PDGF is faster and more efficient than any other generic generator and can be run with perfect speedup in multi-core and multi-node environments. PDGF can write data in various formats (e.g., CSV, JSON, XML, and SQL) to files, database systems, streaming systems, and modern big data storage systems (e.g., HDFS).

The remainder of this paper is structured as follows. We describe PDGF briefly in Section 2 and then show the core functionality of DBSynth in Section 3. After an evaluation in Section 4, we explain our demonstration in Section 5. In Section 6, we give an overview of related work, before concluding in Section 7.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2735378.

2. PDGF

The Parallel Data Generation Framework (PDGF) is a versatile, generic data generator [17]. PDGF has a novel computation-based generation strategy that enables a completely parallel generation of data. PDGF's generation strategy is based on the exploitation of determinism in pseudo random number generators (PRNG). Random number sequences generated using PRNGs are repeatable, which means that the exact same sequence can be generated over and over again. PDGF uses xorshift random number generators, which behave like hash functions. Repeatable, parallel data generation is possible using an elaborate seeding strategy (depicted in Figure 1). In the example, starting with a project seed one seed per table is generated. This seed is used to create one seed per column, which again is used to generate one seed per abstract time unit and finally per field. The field seed is used to generate the random number sequence for the value generation. The values are generated using *field value generators*. These can be simple generators, like number generators, generators based on dictionaries, or reference generators, but also *meta generators*, which can concatenate results from other generators or execute different generators based on certain conditions [18]. The concept of meta generators enables a functional definition of complex values and dependencies using simple building blocks. Although the seeding hierarchy and meta generator stacking seems expensive, most of the seeds can be cached and the cost for generating single values is very low. We will show an analysis of the exact costs in Section 4.

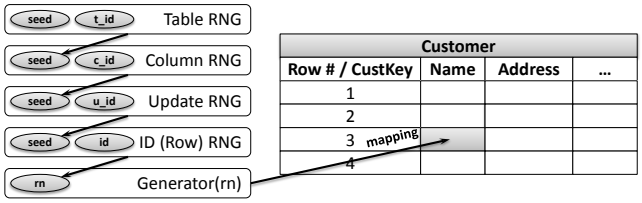

Figure 1: PDGF's seeding strategy

PDGF's architecture is shown in Figure 2. The user specifies two XML configuration files, one for the data model and one for the formatting instructions. These will be explained in more detail in Section 3. Additionally, all previously specified properties of a model and format (e.g., scale factors, table sizes, probabilities) can be changed in the command line interface. The controller then initializes the system. The meta scheduler manages multi-node scheduling, while the scheduler assigns work packages to the workers. A work package is a set of rows of a table that need to be generated. The workers then initialize the correct generators using the seeding system and the update black box. Whenever a work package is generated, it is sent to the output system, where it can be formatted and sorted.

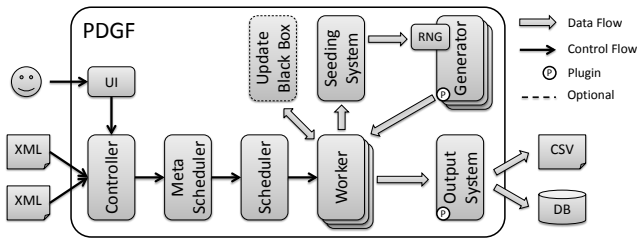

Figure 2: PDGF's architecture

PDGF has been successfully used to implement a variety of benchmarks, e.g., TPC-H [17], the Star Schema Benchmark [19], TPC-DI [6], and BigBench [7].

3. DBSYNTH

DBSynth is an extension to PDGF that automates the configuration and enables the extraction of data model information from an existing database. DBSynth's abstract architecture and mode of operation can be seen in Figure 3. In DBSynth, the user specifies projects, which integrate workflows, such as data generation, data extraction, etc. The figure shows the complete automatic workflow, from model extraction to data generation. Not all steps are necessary for a given project.

DBSynth connects to a source database via JDBC, using the model creation tool, schema information and a configurable level of additional information of the data model are extracted. Possible information includes min/max constraints, histograms, NULL probabilities, as well as statistic information collected by the database system such as histograms. DBSynth also features a rule based system that searches for key words in the schema information and adds predefined generation rules to the data model. For example, numeric columns with name *key* or *id* will be generated with an *ID generator*.

If sampling the database is permissible, the data extraction tool builds histograms and dictionaries of text-valued data and stores the according probabilities for values. Users can specify the amount of data sampled and the sampling strategy. In future versions, we will include a dynamic sampling, which adapts the sample size and sampling strategy according to the base data. If the text data contains multiple words, DBSynth uses a Markov chain generator, which analyzes the word combination frequencies and probabilities. These are stored and linked to the data model.

Using the generated data model, PDGF can generate the data. The model is translated into a SQL schema, which is loaded into the target database using JDBC. The data can be loaded into the target database either using SQL statements generated by PDGF or a bulk load option, if featured by the target database.

Listing 1 presents an excerpt of the automatically generated configuration for a TPC-H data set [15]. The excerpt shows the general structure of the PDGF schema configuration. It contains the project's seed, changing the seed will modify every value of the generated data set, the random number generator, PdgfDefaultRandom is a custom built, very fast xorshift PRNG, property definitions, which can also be changed from the command line, and the schema information itself. A default property is the scaling factor *SF*, which is used to determine the size of the data set. DBSynth will generate a size property for each table and assign it the product of the scale factor and the original table size. This way other scaling dependencies can be easily specified in a centralized point in the model. Furthermore, all boundaries for numerical values and dates are stored in properties.

The schema model is specified in form of table entries. Each table specifies its size, which DBSynth sets to be linear with the scale factor as shown in the example. However, any formula can be used to calculate the size. Then the columns of the table are specified in form of field entries. The first field is "l_orderkey", the name and size are extracted from the database's schema information. The fact that it is a key is deduced from the column name, this is the reason why DBSynth chooses an ID generator for this column. The next field is "l_partkey", which is a reference to the table "partsupp". This is specified in the schema, which is why DBSynth chooses a DefaultReferenceGenerator, which will generate consistent references to this table. The final field that is shown is "l_comment",

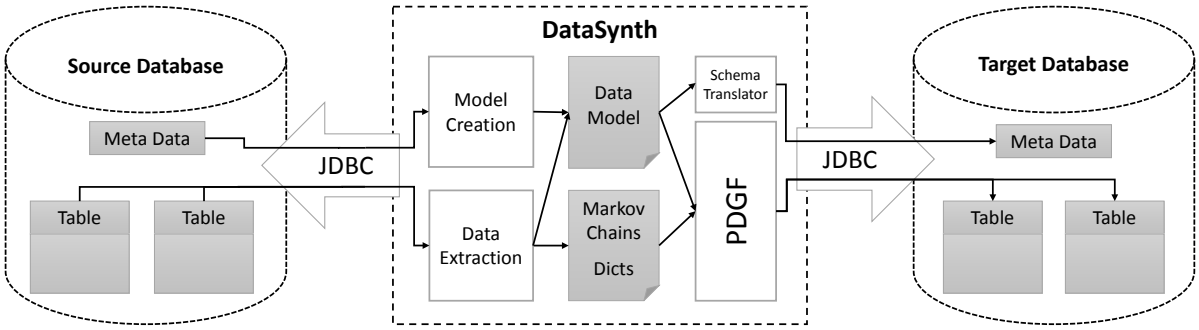

Figure 3: Abstract architecture and data flow in DBSynth

a text field containing free text. DBSynth chooses the Markov-Generator for this field, thus it will sample the original database to build the Markov model. For a TPC-H data set the comment field model contains 1500 words and 95 starting states, which can easily be fit in memory. The choice of the generator type used for a field is based first on referential integrity constraints, i.e., a reference will always be generated by a reference generator independent of its type. Then the data type determines if a number generator, e.g., Long, Integer, Double, or a date generator, or a text generator is used, DBSynth and PDGF support all SQL 92 datatypes. If the database is not sampled, the column name is parsed to determine whether a matching high level generator construct exists, e.g., names, addresses, comment. In case nothing is found a random string is generated. The Markov generator builds dictionaries for single word text fields and Markov chains for free text, the parameters for the Markov model are adjusted based on the original data. If the original data cannot be sampled or analyzed, DBSynth falls back to random values based on the database statistics as well as predefined generators for URLs, addresses, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema name="tpch">
 <seed>12456789</seed>
 <rng name="PdgfDefaultRandom"></rng>
 <property name="SF" type="double">1</property>
 <property name="lineitem_size" type="double">6000000 *
      ${SF}</property>

 <table name="lineitem">
  <size>${lineitem_size}</size>

  <field name="l_orderkey" size="19" type="BIGINT"
       primary="true">
   <gen_IdGenerator>
   </gen_IdGenerator>
  </field>

  <field name="l_partkey" size="19" type="BIGINT"
       primary="false">
   <gen_DefaultReferenceGenerator>
    <reference table="partsupp"
         field="ps_partkey"></reference>
   </gen_DefaultReferenceGenerator>
  </field>
  [..]
  <field name="l_comment" size="44" type="VARCHAR"
       primary="false">
   <gen_NullGenerator> probability=".0000d"
    <gen_MarkovChainGenerator>
     <min>1</min>
     <max>10</max>
     <file>markov\l_comment_markovSamples.bin</file>
    </gen_MarkovChainGenerator>
   </gen_NullGenerator>
  </field>
 </table>
 [..]
```

Listing 1: Excerpt of the schema definition for TPC-H

4. EVALUATION

We evaluated the performance of PDGF and DBSynth on a 24 node, dual socket, dual core cluster and on a single node with two sockets and eight cores per socket. Where possible, generated data was written to */dev/null* to ensure the throughput was not I/O bound. In the experiments, we used either the BigBench data set or our custom implementation of the TPC-H data set.

In the first experiment, we evaluate the performance of PDGF by generating a BigBench data set of scale factor 5000, which results in a total data size of 4392 GB on the 24 node cluster. The results of this experiment can be seen in Figure 4. As is shown in the figure, PDGF has linear throughput scaling in the number of nodes.

In the second experiment, we benchmark the scale out performance of PDGF by increasing the number of workers and thus threads used for the data generation. This experiment is conducted on the single node. The results can be seen in Figure 5. PDGF's throughput increases linearly with the number of cores (16) and further increases with the number of hardware threads (32), but not as significantly as for the number of cores. An interesting observation is that scheduling exactly the same number of workers as the number of system cores or threads is not optimal due to the additional internal scheduling and I/O threads.

In Figure 6, a comparison of the data generator DBGen and PDGF is shown. As can be seen, both tools achieve a similar performance. In parallel mode, it is not possible to write to /dev/null using DBGen, which is why in this experiment the throughput of both, DBGen and PDGF, was disk-bound. We also show PDGF's CPU-bound performance, which is 33% higher than its disk-bound performance. DBGen's parallelization is non-transparent. This means that for each parallel stream a new instance is started, which writes its own files. As a result, DBGen's parallel output will be split in as many files as instances were started, whereas PDGF writes sorted output into a single file. PDGF also supports the same parallel generation strategy as DBGen does, which is starting multiple instances and generating a distinct range of the data set with each instance. With this approach it is possible to scale out the generation to shared nothing systems with linear speedups [17]. When comparing the single process performance, i.e., starting only a single DBGen instance and running PDGF with a single worker, DBGen achieves 48 MB/s and PDGF 30 MB/s. Thus, PDGF is has the

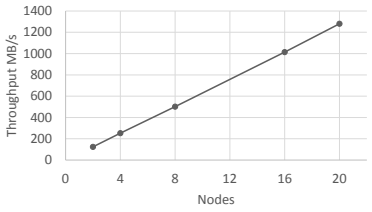

Figure 4: PDGF BigBench scale-out performance

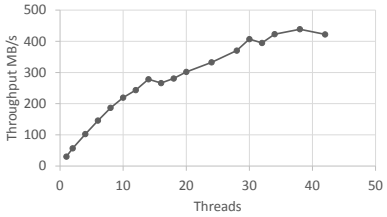

Figure 5: PDGF TPC-H scale-up performance

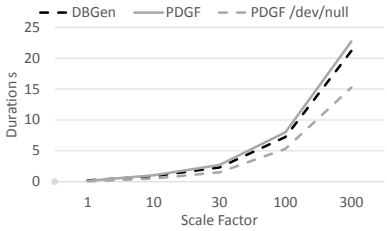

Figure 6: DBGen vs PDGF performance

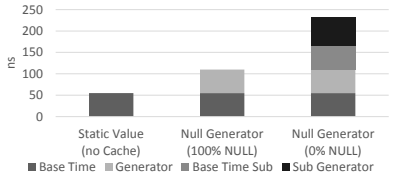

Figure 7: Generation latency

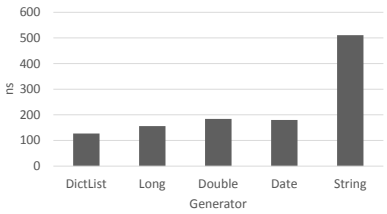

Figure 8: Basic generator latency

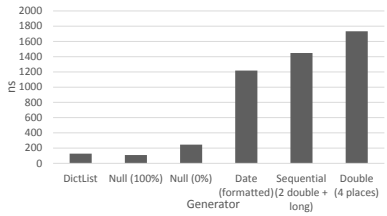

Figure 9: Complex generator latency

same order of performance as DBGen, although being completely generic and adaptable.

We conducted further experiments to determine the sources of latencies for the individual value generation. The experiments were done in a single threaded setup, to get the per value overhead. These results show the pure computational requirements and do not discuss latencies added by the I/O subsystem. In Figure 7, the latency of independent value generation is broken down into its subparts. For a static value, i.e., a column contains only one unique value that is never changed, the pure system overhead can be seen. It is in the order of 50 Nanoseconds (ns). If a NULL value generator is wrapped around a static value that is NULL with 100% probability, the overhead of the NULL generator is added to the generation of the (static) NULL value, this is again in the order of 50 ns. The definition of the NULL value generator can be seen in Listing 1. Finally, if the NULL probability is 0% the inner static value generator has to be executed in all cases, this adds the base time for the subgenerator and the actual value generation, both of which are again ca 50 ns. Thus the total duration for each value is in the order of 200 ns. In Figure 8, it can be seen that this is a good ballpark number for simple values that are not formatted. Picking values from dictionaries, computing random numbers, and generating random strings are all in the range of 100 ns - 500 ns. String formatting is the most expensive operation in data generation in Java, this can seen in Figure 9. Formatting a date value (e.g., "11/30/2014") increases the generation cost to 1200 ns, which is similar to generating a value that consists of a formula that references 2 double values and concatenates it with a long. Although the formatting is expensive, its cost is fixed since PDGF does lazy formatting, which means even very complex values will only be formatted once. This analysis shows that using subgenerators incurs nearly negligible cost (ca. 100 ns) and it also shows that computing values rather than rereading them is much more efficient. While generating complex values might cost up to 2000 ns, doing a single random read will cost ca. 10 ms on disk, which means the computational approach is 5000 times faster than an approach that reads previously generated data to solve dependencies. Furthermore, the computational approach enables a completely parallel generation of data, which is not possible using reading based approach without replicating data or extensive network communication.

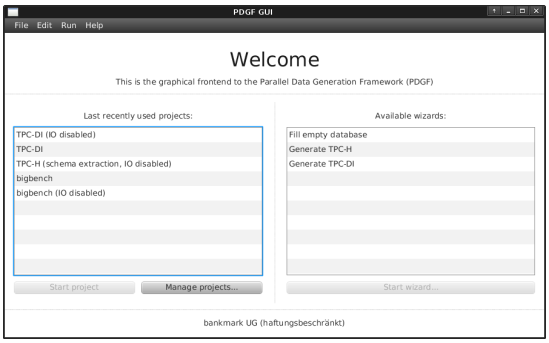

Figure 10: Standard screen of DBSynth

In our final experiment, we tested the performance of the DBSynth metadata extraction. Using a TPC-H database with scale factor 1 loaded in a PostgreSQL DBMS, it takes 600 ms to get the schema information, 1.3 s to get the table sizes, 600 ms to get NULL probabilities, 10 seconds to get all min and max constraints, and between 800 ms (0.001% samples) and 200 s (100% samples) to retrieve data for the Markov chains. These results indicate an interactive response time for data model generation. Using PDGF's preview generation, which shows samples of the generated data instantaneously, data models can be built and improved very fast.

5. DEMONSTRATION

To show the ease of use of DBSynth, we will demonstrate typical work flows for data generation. We will start by generating industry standard data sets such as TPC-H. The data will be generated using PDGF, but this configuration is compliant to the TPC-H data set [15] and was developed in cooperation with the TPC-H subcommittee. This is a default project in DBSynth. The according selection screen can be seen in Figure 10. We will generate a 10 GB TPC-H data set. We will show how the data can be altered

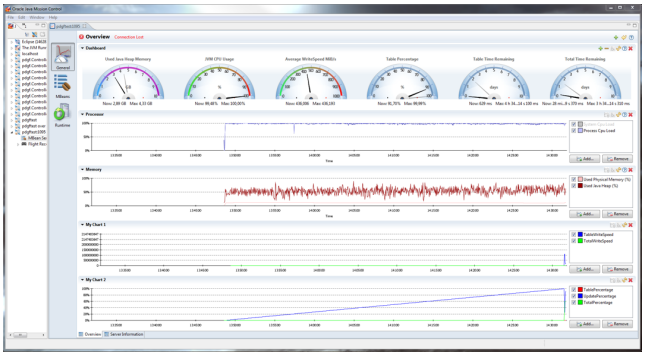

Figure 11: Mission Control interface for PDGF

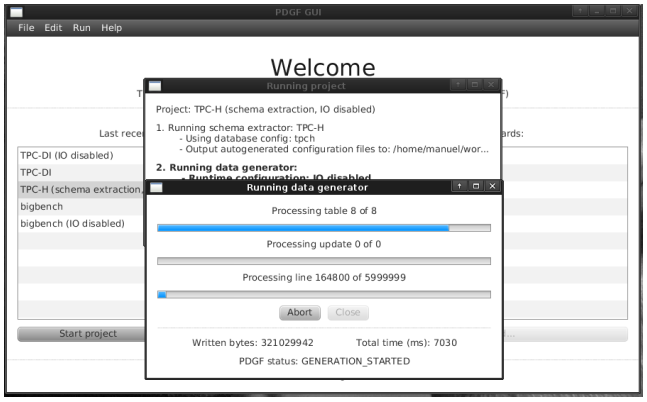

Figure 12: Data generation screen of DBSynth

by changing the output format. To this end, the data will be written in CVS and XML format. The generation progress and system utilization will be monitored using Java Mission Control[1]. PDGF uses the Java Management Extensions internally for inter process communication and using these interfaces, the progress of single tables and the complete data set as well as general performance parameters can be visualized. This can be seen in Figure 11.

After the data generation, we will demonstrate the configuration generation. To show a real use case, we will use the publicly available parts of the IMDb database[2]. The data set is hosted in a MySQL database, which was loaded using the imdbpy2sql.py script, which is part of the IMDbPY package[3]. We will first use a basic schema extraction, where only the schema information is retrieved from the database and no tables are accessed. The generated XML file will be explained, which contains the model for the data generation. Then we will do a second more elaborate schema extraction, where min/max constraints, NULL values, and data samples for Markov chains will be read from the database. We will compare the newly extracted model with the first one and then generate the data. The according screen can be seen in Figure 12. We will show excerpts of the generated data in comparison to the original data and verify the quality by running SQL queries on the original data and the generated data and compare the results. To this end, the generated data will be loaded to a database system.

Finally, we will explain how the model can be changed or adapted. We will change the automatically generated configuration by adding additional columns to the model and refining correlations that could not automatically be detected. This will be done using an automatically generated version of the TPC-H configuration. We will then show the differences between the original TPC-H configuration and the newly generated configuration and compare the generated data sets.

6. RELATED WORK

There is a rich body of work on data generation for database benchmarking and testing. In the following, we will first give an overview of related work on data generation in general and then show other approaches for synthesizing existing data sets.

Even though there are many generic data generation tools, most data generators either produce very simple data or are non-reusable hard coded programs or collections of scripts. Examples for simple data generation are the data generator used by all variations of the sorting benchmark (e.g., TeraSort[4]) and the Yahoo Cloud Serving Benchmark data generator [5]. Examples for hard coded generators are all TPC data generators, with the notable exceptions of TPC-DS [14] and TPC-DI [16]. While simple data is helpful for testing basic functionality it does not represent real world use cases. However, hard coded data generators cannot easily be adapted for changing requirements and for different systems. In many cases real data can be used, but due to the impossiblity of scaling, privacy constraints and cost of storage and transportation it is not feasible for benchmarking.

An important characteristic for benchmarking data is repeatablity. Basis for repeatable parallel data generation is the work by Gray et al. on synthetic data generation [8]. This work describes how to generate non uniform data in parallel on shared nothing systems. These techniques along with parallel pseudo random number generators for the basis of our Parallel Data Generation Framework.

Modern generic data generators can be divided into three subsets according to their reference and correlation generation: (1) no reference generation; (2) reference tracking; and (3) reference computation. Many data generators do not generate references or correlations explicitly, but rely on users providing correct statistic distributions to generate correlating values. Generators that track references either compute all references at the same time that the original value is generated, or they track the original value when a references is generated. The former approach is frequently done using graph models [10, 11] or declarative description [2, 24] which can lead to realistic data, however, typically is very slow and hard to parallelize. Tracking references is done by rereading the previously generated data. This approach was for example presented by Bruno et al. [4], this approach is very flexible but also very slow and does not scale well. A faster approach is generating all related data at the same time. Generic suites that use this approach include MUDD [22] and PSDG [9]. The fastest way of generating correct references in most cases is recomputing them. This approach was first implemented in PDGF. A very similar data generator, Myriad, was built at the Technical University of Berlin [1]. It shares the same generation strategy with PDGF, however, does does not include many of the features of PDGF, such as update generation and text generation.

As part of the BigDataBench suite of test cases for big data systems, an early version of PDGF is used to generate structured data [13]. The suite comprises additional generators for graphs

[1]http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html
[2]http://www.imdb.com/interfaces
[3]https://github.com/alberanid/imdbpy

[4]`https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html`

and a similar text generator as the current version of PDGF features. Unlike PDGF's text generator, the different generators are not connected and, therefore, cannot generate heterogeneous data sets with references in between different data sets, e.g., references from structured data into text.

Although synthetic data is usually better suited for benchmarking purposes, synthetic data should reflect characteristics of real data. Therefore, typically real data sets are analyzed for modeling data sets for benchmarking. Although this step is typically manual, it can be automated. Like DBSynth other tools use the metadata stored in database systems to get information about the distribution and structure of the data. RSGen reads metadata and schema information of existing data sets and generates similar data sets by using histograms of the original data [20]. Although similar to DBSynth, the approach is limited to numerical data. Another tool that is able to scale existing data sets is UpSizeR [23]. It builds a graph of the original schema information and correlation information and generates data accordingly. However, the individual, non-key values are deemed application specific and thus have to be specified by the user. DBSynth uses sampling of the data set to generate dictionaries and Markov chains for non-key, non-numerical values. Furthermore, DBSynth uses its built in dictionaries to increase the value domain in scale out scenarios. Myriad also comes with configuration generation tool Oligos, which can analyze the schema and statistical information of a DB2 database [1]. However, it cannot not sample a database and also has no tools to analyze and synthesize values.

A line of work that is orthogonal to our work is test data generation for queries. QAGen analyzes queries to generate data that produces desired results and intermediate results to cover all required test cases [3]. A similar tool is MyBenchmark [12]. Analyzing queries to ensure desired results is currently not a feature of our tool, but will be included in future versions. Given the deterministic approach of data generation, our tool will then also be able to directly execute the query without ever generating the data, which can be used to verify results for correctness.

Although many of the features of DBSynth are covered to some extent by other projects, none of these does include synthetic value generation. Values are always treated as atomic units. This is problematic for big data sets, where values frequently are texts that have to be further analyzed using machine learning techniques. To generate data sets, which satisfy requirements of big data use cases, DBSynth includes Markov chain generators and dictionaries that can generate realistic, synthetic values based on sampled data.

7. CONCLUSION

We demonstrate DBSynth, an extension to the Parallel Data Generation Framework, which enables a fully automatic configuration of the data generator based on existing databases. DBSynth can build realistic data models from a deployed database extracting schema information, sampling, and analyzing the database. It uses heuristics for data type determination and builds dictionaries and Markov models for data generation. The generated models and configuration can be directly used by PDGF to generate data for a target database. A simple, intuitive graphical user interface ties all parts together and features wizards to guide users through every step of the process. In our demonstration, we showcase the ease of use, the high performance, and the flexibility of the system.

In future work, we will extend DBSynth to automate the complete benchmarking process. To this end, we will generate the queries consistently using PDGF and build additional driver and analysis modules. Furthermore, we will include query analysis to generate data sets with predefined (intermediate) results and generate verification results for queries for given data models.

8. REFERENCES

[1] A. Alexandrov, K. Tzoumas, and V. Markl. Myriad: Scalable and Expressive Data Generation. In *VLDB*, 2012.

[2] A. Arasu, R. Kaushik, and J. Li. Data Generation Using Declarative Constraints. In *SIGMOD*, 2011.

[3] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: Generating Query-aware Test Databases. In *SIGMOD*, 2007.

[4] N. Bruno and S. Chaudhuri. Flexible Database Generators. In *VLDB*, pages 1097–1107, 2005.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154, 2010.

[6] M. Frank, M. Poess, and T. Rabl. Efficient Update Data Generation for DBMS Benchmark. In *ICPE*, 2012.

[7] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: Towards an industry standard benchmark for big data analytics. In *SIGMOD*, 2013.

[8] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*, pages 243–252, 1994.

[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*, pages 1243–1246, 2006.

[11] 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*, pages 707–712, Washington, DC, USA, 2006. IEEE Computer Society.

[12] E. Lo, N. Cheng, and W.-K. Hon. Generating Databases for Query Workloads. *PVLDB*, 3(1-2):848–859, 2010.

[13] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan. BDGS: A Scalable Big Data Generator Suite in Big Data Benchmarking. In *WBDB*, 2013.

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

[15] M. Poess, T. Rabl, M. Frank, and M. Danisch. A PDGF Implementation for TPC-H. In *TPCTC*, 2011.

[16] M. Poess, T. Rabl, H.-A. Jacobsen, and B. Caufield. TPC-DI: The First Industry Benchmark for Data Integration. *PVLDB*, 13(7):1367–1378, 2014.

[17] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A Data Generator for Cloud-Scale Benchmarking. In *TPCTC*, pages 41–56, 2010.

[18] T. Rabl, M. Poess, M. Danisch, and H.-A. Jacobsen. Rapid Development of Data Generators Using Meta Generators in PDGF. In *DBTest*, 2013.

[19] T. Rabl, M. Poess, H.-A. Jacobsen, P. E. O'Neil, and E. O'Neil. Variations of the Star Schema Benchmark to Test Data Skew in Database Management Systems. In *ICPE*, 2013.

[20] E. Shen and L. Antova. Reversing Statistics for Scalable Test Databases Generation. In *DBTest*, 2013.

[21] V. Sikka. Does the World Need a New Benchmark? `http://www.saphana.com/community/blogs/blog/2013/09/16/does-the-world-need-a-new-benchmark`, 2013.

[22] J. M. Stephens and M. Poess. MUDD: a multi-dimensional data generator. In *WOSP*, pages 104–109, 2004.

[23] Y. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Y. Lin, and Y. Lin. UpSizeR: Synthetically Scaling an Empirical Relational Database. *Information Systems*, 38(8):1168–1183, 2013.

[24] E. Torlak. Scalable Test Data Generation from Multidimensional Models. In *FSE*, 2012.