Reversing Statistics for Scalable Test Databases Generation

Entong Shen [*]
North Carolina State University
eshen@ncsu.edu

Lyublena Antova
Pivotal Inc. [†]
lantova@gopivotal.com

ABSTRACT

Testing the performance of database systems is commonly accomplished using synthetic data and workload generators such as TPC-H and TPC-DS. Customer data and workloads are hard to obtain due to their sensitive nature and prohibitively large sizes. As a result, oftentimes the data management systems are not properly tested before releasing, and performance-related bugs are commonly discovered after deployment, when the cost of fixing is very high. In this paper we propose RSGen, an approach to generating datasets out of customer metadata information, including schema, integrity constraints and statistics. RSGen enables generation of data that closely matches the customer environment, and is fast, scalable and extensible.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Design, Performance

Keywords

Data generation, database metadata, parallelization

1. INTRODUCTION

Improving the performance of database systems has been a widely studied topic in the research community. While a number of synthetic benchmarks exist to guide performance tests of database systems in various domains, including TPC-H [8], TPC-DS [7], Star Schema [16], and others, these are far from representing realistic customer use cases. However, often they are the only possibility, as customer data and workloads are hard to obtain, due to their sensitive nature. Moreover, they are usually very large which makes transmission over the network hard if not impossible.

[*]Work is done while author was an intern at Greenplum.
[†]Formerly Greenplum/EMC

Existing synthetic benchmarks for testing the performance of query engines suffer from several limitations:

- **Fixed schema and workloads.** The few available synthetic data generators only offer a fixed schema and workload. While those benchmarks may resemble a certain reporting scenario, they cannot accommodate various schema and workloads in real customer databases, which may have very distinct data characteristics compared to the existing benchmarks.
- **Limited control over the generated datasets.** Synthetic query generators do not offer enough flexibility in generating a variety of datasets. Data size is commonly the only controllable parameter. Some generators like TPC-H allow users to load from a pre-defined set of distributions, such as normal or zipfian, but offer little extensibility, like adding new tables, constraints, and custom distributions.
- **Expensive and impractical.** Recent work [2, 9, 18] has proposed generating workload-aware datasets with the help of constraint solvers. However, these do not scale well to the amounts of data typically present in a customer dataset. See Section 6 for a more detailed discussion on related work.

To overcome the limitations of existing data generators, we propose a mechanism for generating data from database metadata. The metadata is usually less sensitive than the real data and significantly smaller in size: in the order of megabytes compared to multiple terabytes to petabytes commonly seen in customer use cases. Yet, it retains critical information about database schema, integrity constraints and statistics on each column. The idea of 'reversing' the metadata (especially the stats[1]) can benefit the following use cases:

- **Troubleshoot customer issues.** When customers experience functional or performance related problems with the query execution engine, the data generator can speed up resolving the issue in-house without requiring access to the client's production system.
- **Quality assurance using real data.** Testing the performance of a database system on real data and workloads in addition to synthetic benchmarks can greatly improve the robustness of the system. Greenplum and many other database vendors have customers in various industries, whose data and workloads naturally exhibit different characteristics, and may expose corner cases and hidden bugs in the systems.
- **Tune query optimizers.** The cardinality estimator and cost model of a typical query optimizer heavily rely on the statis-

[1]In this paper we use 'statistics' and 'stats' interchangeably.

tics stored in the database catalog. Statistics only *approximate* the real data distribution. Thus, for two databases having the same statistics, the optimizer will likely choose the same query execution plan for a given query. However, they may have different run-time performance when executed on the two instances. RSGen can be extended to generate different databases with the same statistics by augmenting the input with additional parameters that might not be captured in the metadata.

- **Generate datasets based on anonymized metadata.** In some cases, customers may be willing to share an anonymized version of their data to the database vendor. Even in this case, the problem of transmitting the data remains. A solution to that is to anonymize the metadata, which is much smaller in size, and generate datasets from that. There are readily available solutions for the anonymization of metadata and workload, which preserve statistical information, e.g. the ones presented in [6] and in [12].

In this paper, we present RSGen ('RS' stands for 'Reversing Statistics'), a mechanism for automatic generation of datasets based on metadata[2]. RSGen enables generation of datasets truthful to the metadata, as demonstrated in our experimental section. It is designed with speed and scalability in mind – RSGen uses a bucket based model at its core, which is able to generate trillions of records with minimum memory footage. The biggest challenge to the scalability of a data generator is the handling of dependent columns. We introduce a procedure called *bucket alignment* to break the dependency while satisfying referential integrity constraints. Being able to generate dependent columns independently is the basis for the high parallelizability and hence the scalability of our approach. In terms of extensibility, RSGen is agnostic to underlying database system and can easily be extended to work with other systems, as all metadata, such as table definitions, histograms and constraints are translated into system-independent data structures. Moreover, RSGen can be extended to support new types of constraints, histograms etc. Experimental study shows that RSGen is fast, scalable and the generated databases have similar data characteristics as the originals.

The rest of paper is organized as follows. We start off with some preliminaries in Section 2. Section 3 describes the general architecture and four components of RSGen, and Section 4 details on the process of buckets generation and handling referential constraints. Section 5 provides an experimental evaluation of our approach. Finally, we review related work in Section 6 before we conclude.

2. PRELIMINARIES

2.1 Database Statistics and Constraints

Most commercial database systems use statistics to compactly represent data distribution. During query optimization, statistics are used to estimate cardinality of intermediate results, and are a building block in deciding the optimality of a query plan in cost-based query optimizers.

The statistics model of Greenplum Database is based on the one employed by PostgreSQL, see for example [11]. Statistics are collected using the `ANALYZE` command and stored in the system catalog, mainly in the `pg_statistics` table. Each entry in that table represents column-level statistics, and contains among other things the following information:

[2]In this paper, the term *metadata* includes schema information, constraints and statistics.

- Fraction of NULL values
- Number of distinct values
- Average width in bytes
- Equi-depth histogram
- Most common values (MCVs) and their frequencies (MCFs)

In addition, the system catalog stores table-level statistics, including total number of tuples, number of disk blocks, etc. In Greenplum Database, this information is maintained in the `pg_class` table in the system catalog.

Integrity constraints are another piece of metadata used by modern query optimizers to produce the optimal plan. Those typically include unique, foreign key, and check constraints.

Greenplum Database provides several tools, which allow DBAs to extract metadata information from the database, such as `gpsd`, which collects metadata from the system catalog into a '.sql' file, and AMPERe [1], whose output format is XML-based.

2.2 Design Goals

Before we delve into the architecture of RSGen, we discuss the principles and requirements that guided the design of RSGen. Generally speaking, for a data generator to be useful in practice, it must have the following properties:

- **Fast and practical.** The generator should be able to generate large amounts of data in a feasible amount of time. This excludes approaches based on constraint solvers as the latter do not scale well with the number of constraints, and statistics typically generate a large number of constraints. See Section 6 for a more detailed discussion on related work.

- **Scalable.** It is not uncommon to see customer datasets containing trillions of records, and the data generator must scale to support such use cases. Therefore being able to parallelize the data generation is crucial to the scalability of the generator.

- **Support of integrity constraints and common column correlations.** Correlations are present in nearly all datasets and have great influence on the optimal plan choice, and thus system performance. Correlations can either be enforced through integrity constraints, such as uniqueness and referential constraints, or can be inferred from the data and be present in multi-column histograms. Due to their impact on performance, a data generator must take those into account.

- **Truthful.** To best model customer use cases, the generator must preserve the statistical properties in the generated data set, i.e. be truthful to the source data distribution and constraints of the original database.

- **Extensible.** It is desired that the generator can be extended in multiple directions. First, it should be easy to add a new source of input metadata, such as a new database system, or input format. Second, as the database system evolves, the generator must be updated to support improvements in the statistical model. Last but not least, the data generator should provide means to generate multiple datasets for the same input metadata.

3. ARCHITECTURE

Figure 1 provides an overview of the architecture of RSGen. Although RSGen is part of Greenplum's development and test cycles, its design is not bound to any specific database system. At a high level, RSGen takes any kind of metadata dump as the input and generates database tables along with a loading script as the output. It consists of four interconnected modules described next.

3.1 Components

Metadata Store. The metadata store reads and maintains in a system-independent manner various information, which commonly appears in a database catalog such as schema, statistics and constraints. It contains one or more parsers, each of which handles one type of input. Figure 1 shows the current implementation of two parsers, which are able to handle metadata collected by `gpsd` and AMPERe [1]. The parser then converts the metadata into internal data structures of schema, stats and constraints maintained by the metadata store for quick look-up later. For example, the buckets generator (introduced below) may query the metadata store for the data type of a specific column.

Schema Analyzer. The role of the schema analyzer is to interpret and maintain referential integrity (i.e. foreign key constraints) among columns. The schema analyzer builds a referential graph in which each vertex is a column ID and each edge represents a referential constraint. This referential graph is naturally a DAG (directed acyclic graph)[3]. We then topologically sort each connected component to obtain a set of ordered lists of columns, which will be fed into the bucket generator introduced next.

Buckets Generator. At the core of the RSGen database generator is a data model based on a `Bucket` structure defined as follows (constructor omitted):

```
public class Bucket {
    Datum low;
    Datum high;
    long count;
    long nDistinct;
}
```

where `Datum` is an abstract class we defined to represent a data point of any type. The `low` and `high` indicate the lower and upper bound of data values within the bucket, while `count` is the number of data points remain to be generated from the bucket. `nDistinct` specifies the number of distinct values in the bucket. This seemingly simple data structure is in fact quite powerful and versatile in the sense that (1) it unifies common column stats such as null fraction, number of unique values, MCVs, MCFs and histogram – all these stats can be mapped to a set of buckets, see details in Section 4.1); (2) the bucket based data model is central to the massive parallelism design of RSGen. As we will discuss in Section 4.2, a technique called 'bucket alignment' can be used to pre-process the buckets such that referentially constrained columns can be generated independently after the alignment. This is vital for the speed and scalability of RSGen.

Tuple Writer. After the buckets are generated and aligned, the tuple writer materializes the database tables by printing and pushing the tuples to the write buffer. In this process, each `Datum` is mapped back to its original data type and printed accordingly. For example, a value from the `Date` column is internally represented as a `Datum` having a `long` variable and the tuple writer will print out the value using string representation '`MM-DD-YYYY`'. The generated values can be materialized in both column-oriented, as well as row-oriented fashion. For the initial implementation of RSGen we chose the latter, which presents several advantages for us:

[3]Most commercial RDBMSs do not allow circular referencing.

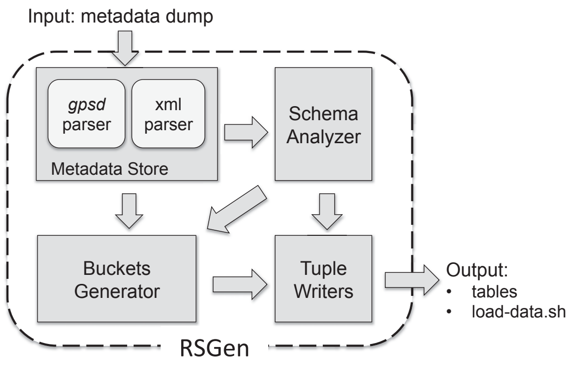


Figure 1: Architecture of RSGen

- the ability to support intra-table constraints and correlations like `CHECK R.a > R.b`
- the generated tuples can be immediately used for parallel data loading, which is available in many commercial RDBMS.

Extending this to a column-oriented approach is straightforward, and has benefits in scenarios where there are not many column correlations, and the target table is column-oriented. A hybrid approach where only dependent columns are materialized together is also easy to support in our framework.

3.2 Scalability and Extensibility

As we mentioned above, the bucket based data model is the key to the scalability of RSGen. Specifically, it enables two layers of parallelism: at the database level, different tables can be generated in parallel after bucket alignment; at the table level, different parts of a table can also be created at the same time. In our implementation of RSGen, the second layer is done by creating multiple workers accessing the available buckets and pushing tuples to a queue, which is then consumed by a tuple writer. Most constraint types commonly found in a database, including key and foreign key constraints, are easily parallelizable by our approach. We acknowledge that other 'exotic' constraints that span across multiple tables and involve arithmetics, for example '$R.t + 0.3 * S.t = W.t$', may require explicitly referencing the generated values. We argue that these are not commonly found in practice. RSGen is able to support intra-table `CHECK` constraints by sacrificing the table level parallelism. We will show in the experiment section that in most cases RSGen is able to scale linearly when the size of the database increases.

RSGen is designed with extensibility in mind from the very beginning. With the separation of functionalities of the four modules, RSGen is able to extend both vertically (supporting new types of constraints) and horizontally (supporting metadata from other RDBMS). Specifically, adapting RSGen to support other RDBMS mainly requires writing a new metadata parser. In addition, if a different type of statistic is used (e.g. MS SQL Server uses MaxDiff histogram instead of equi-depth histogram), the bucket generator only needs to add the functionality of corresponding conforming buckets, while the rest of the data generator remains unchanged.

4. DETAILS

4.1 Buckets Generation

In this part we provide some details of how to generate buckets given the column stats provided by the metadata store. We will

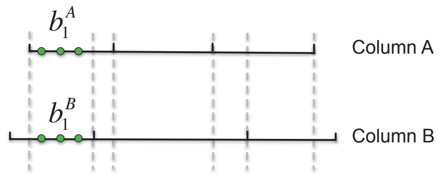


Figure 2: Bucket alignment

focus on the stats available in the Greenplum Database, although extension to other databases is straightforward. Recall from Section 2.1 that the stats for each columns include NULL fraction, number of distinct values, equi-depth histogram, and most common values (MCVs) and their frequencies (MCFs). Given the statistics of a column C and target number of total values and distinct values N and $nDistinct$, respectively, the buckets generator processes the statistics as follows:

1. Process *null fraction* nf:

   ```
   new Bucket (
     low: null,
     high: null,
     count: N*nf,
     nDistinct: 1
     )
   ```

2. Process each *MCV* v with frequency f:

   ```
   new Bucket (
     low: v,
     high: v,
     count: N*f,
     nDistinct: 1
     )
   ```

3. Process *histogram*: the histogram buckets can be directly mapped to the `bucket` structure, by first excluding the MCVs from the histogram. The target number of values and distinct values for this step are obtained by subtracting the values used for the generation of buckets for the NULL values and MCVs from the initial N and $nDistinct$, and distributing those numbers accordingly to each histogram bucket.

4.2 Handling Referential Integrity

As we briefly discussed in Section 1, the biggest challenge in designing a highly scalable data generator is how to generate referential columns in an independent manner without requiring access to what has been generated in the referred column. Reading the referred tuples from disk may easily become the bottleneck of a parallel data generator, and storing them in the memory may be prohibitively expensive[4]. In this part we discuss how our bucket-based data model can bypass this roadblock by bucket alignment. To the best of our knowledge, the only data generator in the literature, which supports generation of dependent data without scanning the referred column's data is [17], in which a multi-layer seeding strategy is used to compute reference. Generally speaking, the strategy we use in RSGen can also be seen as 'computed reference', yet it does not induce high computation cost for generating the keys as in [17].

[4]This does not include the simple case where the referred column is a continuous unique key.

Figure 2 illustrates the process of bucket alignment, showing column A and B each has three histogram buckets initially (solid lines in the plot). Supposing column A refers to column B, the following procedures show how to make sure values generated in column A must exist in column B: (1) the histogram bucket boundaries are aligned, creating a new `Bucket` each time a boundary from either column is encountered. In the example shown in Figure 2, five `Bucket`s are created for column A, and seven for column B after the alignment. When splitting a bucket, the `count` and `nDistinct` are allocated under a uniform assumption, since the distribution within a histogram bucket is unknown. We also require that buckets in the same 'position' (e.g. b_1^A, b_1^B in the figure) have the same `nDistinct`. (2) after the buckets are aligned, the values will be generated from each bucket in a deterministic manner. We first uniformly divide a bucket into `nDistinct` intervals, then a fixed point (e.g. the middle point) from the $(c \ \% \ nDistinct)$-th interval will be generated when the bucket has a current `count` of c. Since buckets in the same position have the same `nDistinct` value, in this way the possible values generated in both columns will be the same. We acknowledge that ideally the values generated from a bucket would be uniformly distributed. By breaking a bucket into intervals and selecting a fixed point for each, we have sacrificed the uniform assumption locally in order to obtain independence and scalability in data generation. This restriction can be overcome by splitting each bucket into sub-buckets to obtain the desired resolution.

5. PERFORMANCE EVALUATION

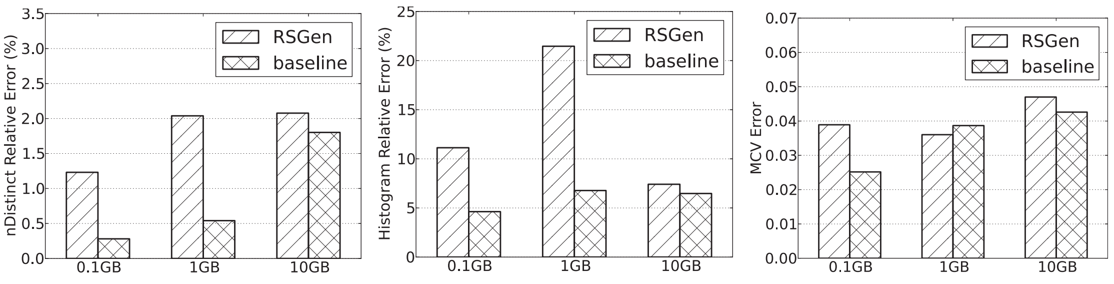
In this section we evaluate the performance of RSGen through extensive experimental study. We first investigate the run time performance, followed by RSGen's ability to recover database statistics. We also test the truthfulness of the generated database by measuring the relative error in range count queries. Finally we discuss the performance of complex queries on database generated by RSGen.

5.1 Setup

All experiments are conducted on a server with a 1.8GHz Quad CPU and 8G of memory. The single node edition of Greenplum Database 4.2.2 is used and configured with 1 master and 2 segment nodes. We use TPC-H benchmark [8] with various scale factors as our test databases. We do not report on experiments with a real customer database here, as the legal and privacy implication prevent us from publishing such results. Instead, we use TPC-H generated by *dbgen*, the native data generator of TPC-H, as the original database, which we then compare to the database generated by RSGen. RSGen was implemented in Java for maximum portability and all results are the average of five runs.

5.2 Performance and Scalability

First we examine the run time performance of RSGen. Figure 3 shows the time of generating the TPC-H benchmark datasets with various scale factors using *dbgen* and RSGen. It can be observed that the two generators have comparable runtime, with RSGen being slightly faster for scale factor 0.1 and slightly slower at larger scale factors than *dbgen*. This performance is reasonable since RSGen is written in Java while *dbgen* is a specialized C implementation. In terms of scalability, RSGen has exhibited the ability to scale up linearly. Note that the above result is obtained in a single node environment – the speed of RSGen is expected to significantly improve in a cluster given its design for parallelism discussed in Section 3.2.



(a) Recovery of nDistinct (b) Recovery of histogram (c) Recovery of MCVs

Figure 4: Recovery of database statistics

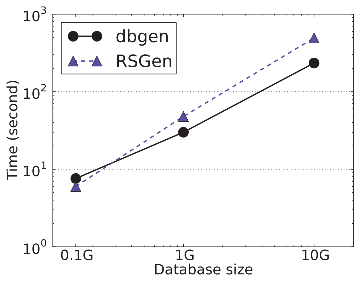


Figure 3: Runtime performance of RSGen

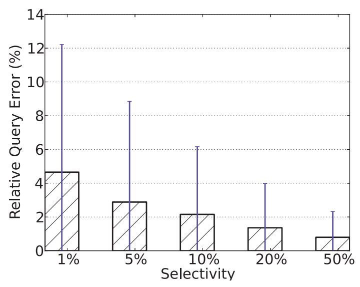


Figure 5: Accuracy of COUNT queries

5.3 Recovery of Statistics

As pointed in Section 2.2, one of the design goal of our data generator is the ability to recover the data characteristics of the original database. In this part, we investigate the closeness of the statistics between the original and generated databases. This is done by directly comparing the stats collected by `gpsd`, Greenplum's stats dump utility, from both databases. First, we define the following metrics for three important types of column statistics in Greenplum Database:

- *nDistinct Error* is defined as the average relative error of *nDistinct* across all columns of the generated database as compared to the original database.
- *Histogram Error* is defined as the average relative displacement of histogram boundaries, i.e., $\frac{1}{n}\sum_{i=1}^{n}|t'_i - t_i|/(t_i - t_{i-1})$, where t_i and t'_i are the ith histogram boundary of the original and generated databases respectively.
- *MCV Error.* To measure the error of recovering MCVs, we must take both the most common values and their frequencies into account. The scenario where multiple MCVs have the same frequency but some of them are not included in the statistics should also be considered. Formally,

$$\begin{aligned}&\textstyle\sum_{s\in\{MCV\cap MCV'\}}|f_s - f_{s'}|\\&+\textstyle\sum_{s\in\{MCV\setminus MCV'\}, f_s>\min(MCF)} f_s\\&+\textstyle\sum_{s\in\{MCV'\setminus MCV\}, f_s>\min(MCF')} f_s\end{aligned}$$

  where f_s and $f_{s'}$ are the frequencies of common values s in the original database and s' in the generated database respectively.

Figure 4 shows the accuracy of the statistics in the generated databases with various scale factors averaged over all columns where corresponding stats are available. Since the collection of database stats is based on sampling, they vary each time the columns are `ANALYZE`d. We therefore establish a baseline of achievable minimum error by running `ANALYZE` on the original database multiple times and measuring the average error. As can be seen in Figure 4, the recovered *nDistinct* is able to achieve a relative error of less than 3% and the recovered *MCVs* are comparable to the baseline. The recovered histogram has less than 10% of relative error except the case of 1G, where a spike is observed due to a large error in one column. Overall we are able to obtain similar stats from the RSGen generated database and accuracy of recovery is able to hold when the size of the database scales up.

5.4 Range Queries

The accuracy of range COUNT queries is a pragmatic metric of the truthfulness of the generated database. Such queries can essentially describe the data distribution of each column. In this set of experiment, we issue range queries of different selectivity (from 1% to 50%) on both databases (0.1GB) and measure the relative error. We generate 20 queries for each column and find that the average errors are less than 5% and decreasing as selectivity increases, as shown in Figure 5.

5.5 Complex SQL Queries

As mentioned in Section 1, one of the motivations for RSGen is tuning query optimizers by measuring how sensitive they are to variations in the input data, which cannot be correctly captured by the statistics capabilities of the source system. In this part we perform experiments with queries from the TPC-H query set, which go beyond the simple independent range queries. Again, we compared the cardinalities of the query results over the original and the generated database. For lack of space we do not include detailed plots of the results. Our observations can be summarized in the following:

- Several TPC-H queries showed discrepancies in the result cardinality.

- Some of these can be explained with known limitations of the PostgreSQL statistical model, including handling text data and column correlations.

- While the actual output cardinalities were different, the query optimizer chose identical plans for the original and the generated database, since both databases agreed on metadata and statistics. These results are valuable, as they model realistic use cases, where two datasets map to the same stats, but may exhibit different query performance. The results can be used to guide improvements in the statistical model and cost computation in the query optimizer.

6. RELATED WORK

There have been two main lines of work in the literature of test database generation. The first one (see [5, 10, 13, 14, 17]) focuses on scalable and parallel generation of large synthetic databases, subject to user-provided closed-form column distribution. An early work by Gray *et al.* [10] discussed congruential generators to obtain dense unique uniform distribution and its extension to other common types of distributions, in which opportunities for scale-up is explored. Many recent works use descriptive languages [5, 13, 17] or a graph model [14] for the definitions of data dependencies and column distributions. In particular, a hierarchical seeding approach is proposed in [17] such that the data generating procedure can be stateless across multiple hosts. Most of these data generators require excessive efforts from the user to learn the descriptive language and accurately specify data dependencies and distributions.

A major issue of using closed-form synthetic data distribution is that the generated database tends to give empty result over complex queries. The reason is that the subtle correlations between attributes are often not captured. Another line of work [2, 3, 4, 15, 9, 18] addresses this problem by considering a richer set of constraints, e.g., generating a database given a workload of queries such that each intermediate result has a certain size. They constraints are typically specified in a declarative language and the use of constraint solvers is very common in these works. For example, Arasu *et al.* [2] identify the declarative property of cardinality constraints and its ability to specify data characteristics. Given a large number of cardinality constraints as input, the paper proposed algorithms based on LP solver and graphical models to instantiate tables that satisfy those constraints.

7. CONCLUDING REMARKS

In this paper we presented RSGen, a data generator that can reverse database metadata, especially the statistics, to generate a synthetic database with similar data characteristics. We have explored the design space and requirements and proposed a bucket-based data model which is able to break the dependencies of referential integrity and provide high scalability. RSGen provides a practical and extensible tool for scalable test database generation that is easy to use by developers and QA engineers. We plan to extend RSGen in the future with additional configuration options to support generation of databases with different properties not captured by the statistical model of the source system, and combine RSGen with data anonymization tools.

Acknowledgments. The authors would like to thank Sivaramakrishnan Narayanan, Florian Waas, and the members of the Query Processing team at Greenplum for their valuable feedback during the project.

8. REFERENCES

[1] L. Antova, K. Krikellas, and F. M. Waas. Automatic capture of minimal, portable, and executable bug repros using ampere. In *DBTest*, page 2, 2012.

[2] A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 685–696, 2011.

[3] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*, pages 506–515, 2007.

[4] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 341–352. ACM, 2007.

[5] N. Bruno and S. Chaudhuri. Flexible database generators. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1097–1107. VLDB Endowment, 2005.

[6] M. Castellanos, B. Zhang, I. Jimenez, P. Ruiz, M. Durazo, U. Dayal, and L. Jow. Data desensitization of customer data for use in optimizer performance experiments. In *ICDE*, pages 1081–1092, 2010.

[7] T. P. P. Council. TPC-DS Benchmark. In `http://www.tpc.org/tpcds/`, 2001 - 2013.

[8] T. P. P. Council. TPC-H Benchmark. In `http://www.tpc.org/tpch/`, 2001 - 2013.

[9] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 67–74, New York, NY, USA, 2010. ACM.

[10] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 243–252, 1994.

[11] T. P. G. D. Group. PostgresSQL Manual. In `http://www.postgresql.org/`, 1996 - 2013.

[12] V. Gupta, G. Miklau, and N. Polyzotis. Private database synthesis for outsourced system evaluation. In *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2011.

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

[14] K. Houkjær, K. Torp, and R. Wind. Simple and realistic data generation. In *VLDB*, pages 1243–1246, 2006.

[15] E. Lo, N. Cheng, and W.-K. Hon. Generating databases for query workloads. *Proc. VLDB Endow.*, 3(1-2):848–859, Sept. 2010.

[16] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In R. O. Nambiar and M. Poess, editors, *TPCTC*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009.

[17] T. Rabl, M. Frank, H. Sergieh, and H. Kosch. A data generator for cloud-scale benchmarking. *Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 41–56, 2011.

[18] E. Torlak. Scalable test data generation from multidimensional models. In *SIGSOFT FSE*, page 36, 2012.