DataSynth: Generating Synthetic Data using Declarative Constraints

Arvind Arasu
Microsoft Research
Redmond, WA
arvinda@microsoft.com

Raghav Kaushik
Microsoft Research
Redmond, WA
skaushi@microsoft.com

Jian Li
University of Maryland
College Park, MD
lijian@cs.umd.edu

ABSTRACT

A variety of scenarios such as database system and application testing, data masking, and benchmarking require synthetic database instances, often having complex data characteristics. We present *DataSynth*, a flexible tool for generating synthetic databases. DataSynth uses a simple and powerful declarative abstraction based on *cardinality constraints* to specify data characteristics, and uses sophisticated algorithms to efficiently generate database instances satisfying the specified characteristics. The demo will showcase various features of DataSynth using two real-world data generation scenarios.

1. INTRODUCTION

We propose to demonstrate *DataSynth*, a tool for generating synthetic database instances. The need for synthetic data arises in a variety of scenarios:

1. *DBMS Testing:* When we design a new DBMS component such as a new join operator or a new memory manager, we require synthetic database instances with specific characteristics to test correctness and performance of the new component [3, 8]. For example, to test the code module of a hybrid hash join that handles spills to disk, we might need a database instance with a high skew on the outer join attribute.

2. *Data masking and database application testing*: Organizations sometimes outsource the testing of their database applications to other organizations. However an outsourcing organization might not be able to share its internal databases (over which the applications run) with the testing organization due to privacy considerations, requiring us to generate a masked database that behaves like the original database for the purposes of testing.

3. *Benchmarking*: In order to decide between multiple competing data management solutions, a customer might be interested in *benchmarking* the solutions [8]. The standard benchmarks such as TPC-H might not capture many of the application scenarios and data characteristics of interest to the customer, motivating the need for synthetic data generation. A related scenario is *upscaling*, where we are interested in generating a synthetic database that is an upscaled version of an existing database. Upscaling is useful for future capacity planning purposes.

Proceedings of the VLDB Endowment, Vol. 4, No. 12

Data Characteristics: To be useful and meaningful for applications, synthetic databases often need to have a variety of data characteristics. A natural class of characteristics are schema properties such as key and referential integrity constraints, functional dependencies, and domain constraints (e.g., *age* is an integer between 0 and 120). A synthetic database for DB application testing often needs to satisfy such constraints since the application being tested might require these constraints for correct functioning. If DB application testing involves a visual component with a tester entering values in fields of a form, the synthetic database might need to satisfy *naturalness* properties, e.g., the values in an address field should "look like" real addresses.

In benchmarking and DBMS testing, we typically need to capture characteristics that can influence the performance of a query over the generated database. These include, for example, ensuring that values in a column be distributed in a particular way, ensuring that values in a column have a certain skew, or ensuring that two or more columns are *correlated*. We note correlations can involve joining multiple tables. For example, in a customer-product-order database, we might need to capture correlations between the age of customers and the category of products they purchase. In data masking, we might require synthetic data to result in the same application performance as the original data, without revealing sensitive information from the original data.

Cardinality Constraints: Most prior approaches to data generation are *procedural* [3, 5, 7, 9, 10]. These approaches provide some basic procedural primitives and a programmer combines these primitives to design a program that outputs a synthetic database. (Some of these approaches [5, 9] are visual tools that allow an user to configure using a menu of pre-programmed options. We classify them as procedural since they are not declarative.) Even with the right procedural primitives, designing a program to realize synthetic data with complex characteristics can be difficult: As a concrete example, consider generating a customer-product-order database where we need to capture correlations between several pairs of columns such as customer age and product category, customer age and income, and product category and supplier location.

In recent work [1] we explore *declarative* approaches to synthetic data generation and we argue that *cardinality constraints* represent a natural, expressive, and declarative mechanism for specifying complex data characteristics. We also present efficient algorithms that can handle large number of constraints and scale well in the size of the generated data. The DataSynth tool is based on the techniques presented in [1]; Section 2 provides a brief overview of these techniques.

MyBenchmark [8] is closely related to our work and also uses cardinality constraints, but has limitations in functionality and performance that we elaborate in [1]. Briefly, MyBenchmark produces as output a small number of database instances that collectively "cover" all constraints. In other words, MyBenchmark is not guaranteed to produce a single database instance and this functionality can be unsuitable for some applications requiring synthetic data. For example, we can not use multiple database instances for DB application testing, since no single instance reflects all the characteristics of the original database.

2. DATASYNTH: OVERVIEW

DataSynth is a visual data generation tool based on the techniques presented in [1]. In this section, we provide a brief overview of these techniques and describe the end-to-end functionality provided by DataSynth.

2.1 Cardinality Constraints

Let $\mathcal{D}$ denote the database being generated and let $R_1, \ldots, R_l$ denote the relations in the database. A cardinality constraint is of the form:

$$|\pi_{\mathcal{A}}\sigma_P(R_{i_1} \bowtie \cdots \bowtie R_{i_p})| = k$$

where $\mathcal{A}$ is a set of attributes, P is a selection predicate, and k is a non-negative integer. A database instance *satisfies* a cardinality constraint if evaluating the relational expression over the instance produces k tuples in the output. More generally, constraints can also involve non-equality operators such as $\leq, \geq, <$, and $>$.

We can customize data characteristics using a set of cardinality constraints and requiring the generated database satisfy all the constraints in the set. For example, we can specify that a set of attributes $\mathcal{A}_k$ is a key for relation R using two constraints $|\pi_{\mathcal{A}_k}(R)| = N$ and $|R| = N$, where π is duplicate eliminating. We can approximately capture the value distribution of a column using a histogram. We can specify a single dimensional histogram by including one constraint for each histogram bucket. The constraint corresponding to the bucket with boundaries $[l, h]$ having k tuples is $|\sigma_{l \leq A \leq h}(R)| = k$. We can capture correlations between attributes using multi-dimensional histograms such as STHoles [4], which can again be encoded using one constraint for each histogram bucket. We can approximately constrain the performance of a query plan over generated data by specifying intermediate cardinalities as shown in Figure 1. Each intermediate cardinality maps to a cardinality constraint. In the data masking scenario from Section 1, these intermediate cardinalities can be obtained by evaluating the query plan on the original data to ensure that the performance of the plan on original and synthetic data are similar. We refer the reader to [1] for a detailed discussion of pros and cons of cardinality constraints for data generation.

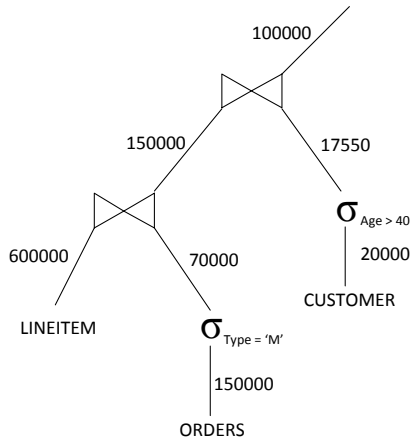

Figure 1: Query Plan intermediate cardinalities.

2.2 Algorithms

The formal data generation problem (DGP) is to identify a database instance that satisfies an input set of cardinality constraints $C_1, \ldots, C_m$. The DGP problem is NEXP-complete, but there exist efficient probabilistically approximate algorithms for a large and useful class of constraints [1]. The following example illustrates some ideas underlying the algorithms.

EXAMPLE 1. *Consider a DGP instance involving a single table R with a single attribute A and the following three constraints: $|\sigma_{20 \leq A < 60}(R)| = 30$, $|\sigma_{40 \leq A < 101}(R)| = 40$, and $|R| = 50$. Assume the domain of A is $[1, \ldots, 100]$. Using the constraints we can identify 4 "relevant" intervals: $[1, 20)$, $[20, 40)$, $[40, 60)$, $[60, 101)$. For each interval $[l, h)$, we introduce a variable $x_{[l,h)}$ to represent the number of tuples in R with A values in the interval. The three constraints can be expressed using the following (integer) linear program (LP).*

$$\begin{aligned} x_{[1,20)} + x_{[20,40)} + x_{[40,60)} + x_{[60,101)} &= 50 \\ x_{[20,40)} + x_{[40,60)} &= 30 \\ x_{[40,60)} + x_{[60,101)} &= 40 \end{aligned}$$

One solution to the LP is $x_{[1,20)} = 2$, $x_{[20,40)} = 8$, $x_{[40,60)} = 22$, and $x_{[60,101)} = 18$. To generate $R(A)$, we pick 2 values (e.g., at random) from $[1, 20)$, 8 values from $[20, 40)$, etc.

A straightforward generalization of this idea to multiple attributes is exponential in the number of attributes and therefore inefficient. The algorithms presented in [1] go beyond simple generalization of the above approach and use "factorization" ideas from probabilistic graphical models for a more efficient solution. An important feature of our algorithms is that they are based on solving an LP; in contrast, the algorithms of [2, 8] rely on general purpose constraint solvers, which are less efficient than LP solvers.

2.3 DataSynth Tool Functionality

We now briefly describe the end-to-end functionality supported by DataSynth. Data generation using DataSynth involves a configuration step where the user configures a data generation task, and a data generation step, where the actual data generation takes place.

Configuration: To configure a data generation task, the user provides the schema of the output database and any primary and foreign key constraints that need to hold in the output. For each column in the schema, the user can specify a domain. A domain is a range of values for linear data types such as *integer* and *date*. For textual data types, the domain can be an explicit enumeration of values, a regular expression, or an implicit enumeration: the user points to a column of an existing table and the set of distinct values in the column comprise the values in the enumeration.

For many data generation applications, there exists a natural *reference database* that forms the basis for generating the synthetic database. For example, in data masking, the synthetic database is a masked version of an original (reference) database; in upscaling, the synthetic database is a scaled version of an original database. DataSynth allows the user to specify a reference database to make configuration simpler. In particular, the user can specify that the schema information (including primary and foreign keys) for the generated database be copied from the reference database.

As discussed above, the user can provide cardinality constraints to customize the data generation. The user can provide cardinality constraints manually by specifying a query expression and cardinality. The tool also allows the user to specify the distribution of values for certain column types and some limited form of two dimensional correlations using a visual interface. These configurations are internally converted to cardinality constraints.

When a reference database is defined, the user can specify more complex constraints in a semi-automated fashion. The user can use the tool to learn single- and multi-dimensional histograms from corresponding columns in the reference database and use them to constrain values in the generated database. We are currently working on another utility, which would allow the user to specify a workload of queries, identify intermediate cardinalities from query plans in the workload (e.g., Figure 1) and use these as constraints. As mentioned earlier, this functionality is useful in data masking to generate synthetic data that has same application performance as original data. (We have not fully explored the privacy related issues in this setting. We note that cardinality constraints integrate nicely with *differential privacy* [6]; instead of using actual intermediate cardinalities, we can fudge cardinalities using differential privacy algorithms and use the fudged constraints for data generation. We are currently exploring these directions.) At any given point in time, the user can view the current set of cardinality constraints as shown in Figure 2.

Data Generation: At any given point in time during configuration, a sample of the generated data based on the current set of constraints is shown to the user. This provides immediate feedback to the user which the user can use to tweak constraints and change configuration. This feature is shown in Figure 3: The top pane shows the original data from a reference database table and the bottom pane shows partially generated data. Once the user is satisfied with the configuration, she can use the tool to materialize the full synthetic database to a back-end DBMS.

3. DEMONSTRATION CONTENT

This section discusses two data generation scenarios we plan to demonstrate.

3.1 Simple Data Generation Scenario

Here, we plan to demonstrate the basic functionality of DataSynth. In particular, we plan to show how using a few configuration settings a user can generate a TPC-H like data. We will also demonstrate how the user can easily introduce skew in column values, change the scale of generated data, and make other customizations by adding a few constraints and changing some configuration settings.

The scenario is interactive and an audience member can play around with the tool and generate synthetic data with various characteristics interactively. This interactive scenario should also give the audience member a sense for the efficiency of the underlying algorithms.

3.2 Data Masking Scenario

Here, we plan to demonstrate a data masking scenario using real-world data and workload. Briefly, the goal of the demo scenario is to generate a synthetic instance of data that has similar performance characteristics as original data for the given workload.

We start off with an existing database instance and identify the subset of columns that need to be masked. To preserve performance of a workload, we identify various intermediate cardinalities of the queries in the workload using the utility mentioned earlier and provide these as constraints to DataSynth. We finally generate a synthetic database instance that satisfies these constraints and we will show (by running some queries) how performance of the queries in the workload are similar for both the synthetic and original database. Some parts of this scenario will be pre-computed off-line to keep the overall demo short.

4. REFERENCES

[1] A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In *SIGMOD*, 2011. To Appear.

[2] C. Binnig, D. Kossmann, E. Lo, et al. QAGen: generating query-aware test databases. In *SIGMOD*, pages 341–352, 2007.

[3] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.

[4] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *SIGMOD*, pages 211–222, 2001.

[5] DTM data generator. `http://www.sqledit.com/dg/`.

[6] C. Dwork. Differential privacy. In *ICALP (2)*, pages 1–12, 2006.

[7] J. Gray, P. Sundaresan, S. Englert, et al. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[8] E. Lo, N. Cheng, and W.-K. Hon. Generating databases for query workloads. In *VLDB*, pages 848–859, 2010.

[9] Red gate sql data generator. `http://www.red-gate.com/products/sql-development/sql-data-generator/`.

[10] J. M. Stephens and M. Poess. MUDD: a multi-dimensional data generator. In *Proc. of the 4th Intl. workshop on Software and Performance*, pages 104–109, 2004.

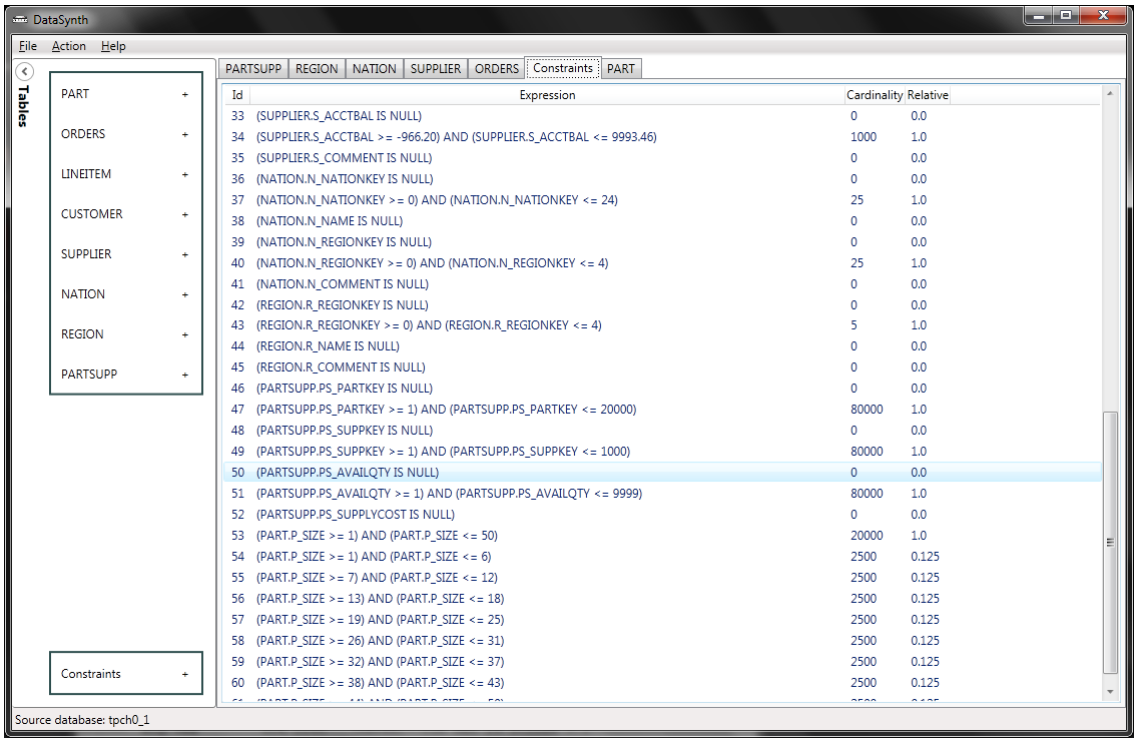

Figure 2: DataSynth: Constraints View.

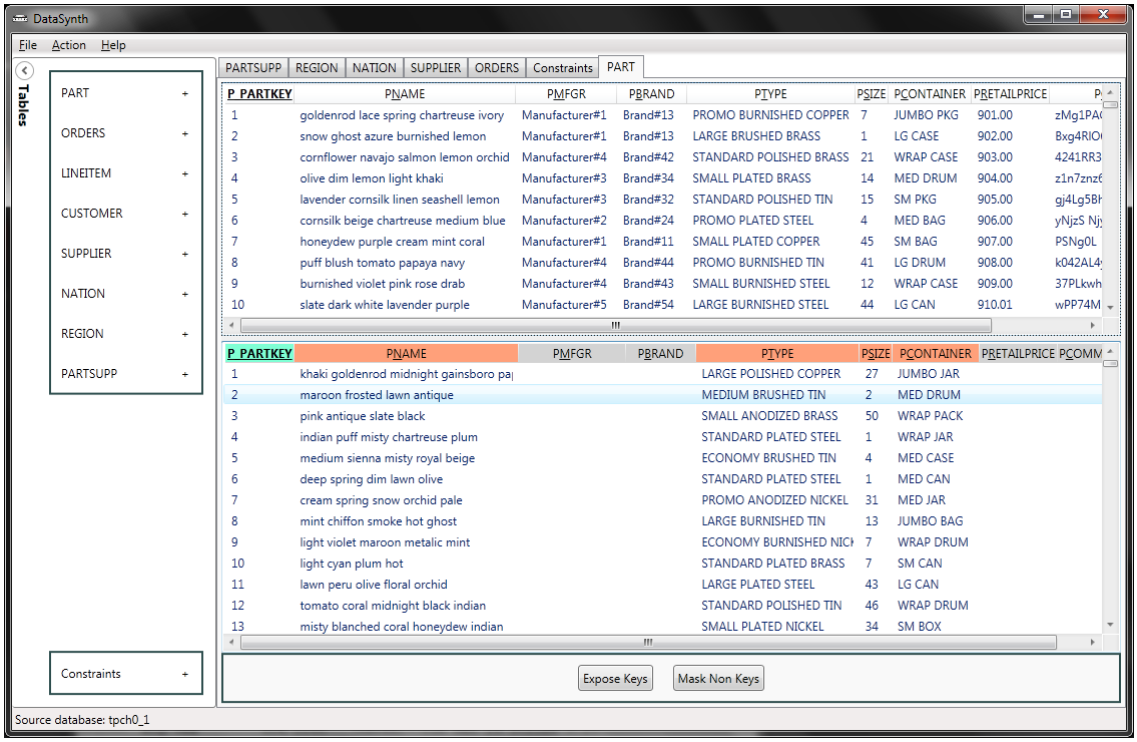

Figure 3: DataSynth: DataMasking View.