Simple and Realistic Data Generation

Kenneth Houkjær
Aalborg University
khp1412@cs.aau.dk

Kristian Torp
Aalborg University
torp@cs.aau.dk

Rico Wind
Aalborg University
rw@cs.aau.dk

ABSTRACT

This paper presents a generic, DBMS independent, and highly extensible relational data generation tool. The tool can efficiently generate realistic test data for OLTP, OLAP, and data streaming applications. The tool uses a graph model to direct the data generation. This model makes it very simple to generate data even for large database schemas with complex inter- and intra table relationships. The model also makes it possible to generate data with very accurate characteristics.

1. INTRODUCTION

Most software developers and database researchers use synthetic data to test the correctness and performance of their work. Such test data must be realistic and correct in terms of size and distributions to be useful, e.g., see the TPC benchmarks [1]. The tools used to generate synthetic data are often specialized and not reusable. This makes the task of building such tools complicated, time consuming, and error-prone. In addition, the specialized tools have limited support for complex database schemas, e.g., composite keys and cycles in foreign-key relationships. This paper presents a simple, generic, and extensible tool for generating synthetic test data. The tool is highly customizable and easy to use. It can significantly reduce the complexity and the time needed to generate realistic and correct test data. The tool can be downloaded from [2].

The demonstration of the tool will focus on the following:

- Presentation of the graph-model based data-generation algorithm, including how the tool automatically builds this graph model.
- How the user can adjust the graph model, including adding additional (primary/unique/foreign) keys to the graph model.
- How the user can very accurately control the characteristics of the generated data, e.g., to evaluate a query optimizer.
- How to handle conflicting user adjustments, e.g., if the cardinality distribution of a foreign key is too high compared to the number of rows in a table.
- How to enhance the realism of the test data by using a non-empty database schema as the outset for the data generation.
- How to increase the data realism by adding new user-defined data types and distributions.
- How to generate data for well-known benchmarks including Wisconsin, TPC-C, and TPC-H.
- How to generate "dirty" data, e.g., to test data-warehouse ETL functionality.

1.1 Example

Figure 1 shows a database schema that is used as a running example in the paper. Boxes in the figure are tables and arrows are foreign-key constraints. A star to the left of a column marks that it is a part of a primary key. The schema consists of four tables and models a software company with employees and projects. The *Employee* and *Project* tables store information on employees and projects, respectively. The *Works_On* table models a many-to-many relationship between the employees and projects. The *Employee* table has a one-to-many relationship to itself that is implemented via the *Works_For_FK* column. There is a one-to-many relationship between the *Project* and *Project_Detail* tables. Please note the composite and overlapping primary and foreign keys.

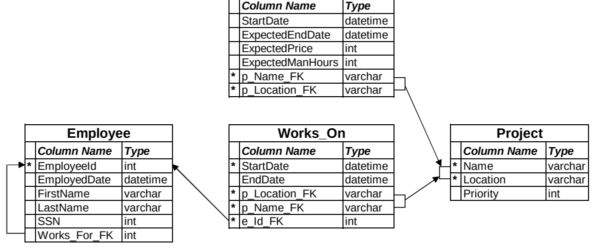

Figure 1: Example Database Schema

2. ARCHITECTURE

Figure 2 shows the architecture of the data generation tool. It consists of a kernel, five extensible component collections, a GUI, and a distributed generation component. The kernel contains the core functionality of the tool and is responsible for the primary, unique, and foreign-key handling. The actual data generation is done by the data types. The kernel and the data types use the distributions to control the data characteristics. The input to the tool is via the meta-data interface. This interface makes the tool DBMS independent and it supports the Oracle, SQL Server, PostgreSQL, and MySQL DBMSs. Additional DBMSs can easily be supported. The tool can output data in several formats including to file, to database, and as streaming data. The user can easily add new data types, outputs, functions, and distributions.

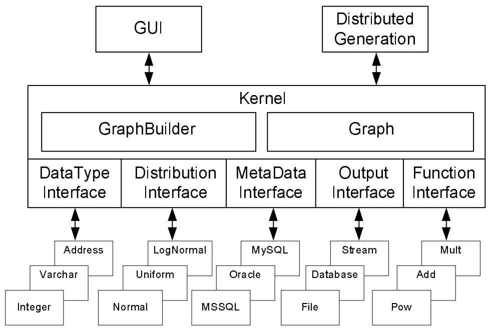

Figure 2: System Architecture

2.1 The Graph Model

Figure 3 shows a directed graph model of the database schema in Figure 1. This model controls the overall data generation. In the model tables are represented as nodes and foreign-key constraints as edges. The model is automatically built using the metadata interface. The letters above each edge indicate whether the edge is a *Normal*, *Forward*, or *Backward* edge [6]. The number of in-bound edges for each node is shown as the number below the node.

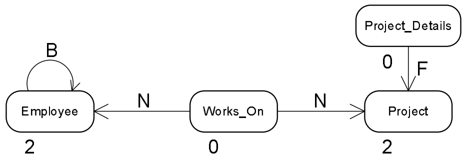

Figure 3: Graph Model

Each node in the graph contains information about a single table. This includes information about columns and primary, unique, and foreign keys. Keys can be overlapping, e.g., a foreign key can be part of a primary key. Composite primary, unique, and foreign keys are supported.

The direction of the edges is from the table containing the foreign key, to the table containing the primary (or unique) key. The edges are used to determine the order in which to generate data.

The graph model holds various statistical information about foreign key and column content. Each edge contains information about a foreign key, including a distribution of the cardinality. The distribution itself and all distribution variables can be adjusted by the user. If the source database schema contains data, the distribution can be based on this data. If no data exist, a default cardinality will be used. The edges also contain information about participation, i.e., how large a percentage of the unique or primary keys that are referenced. The participation can be adjusted by the user.

The number of rows to generate for each table can be set by the user at the database and table level. If the size is set at the database level, all tables are scaled equally, i.e., the number of rows in each table is increased by the same percentage. If the size is set at the table level, the user can choose to propagate the new size, i.e., the size of all related tables (by edges in the model) are updated recursively. It is possible to fix the size of a table and hereby specifying that the table does not allow propagation. This freezes the size of the fixed and possibly related tables. As an example, if the *Project* table is fixed, changes to the size of the *Employee* table will only affect the *Employee* and *Works_On* tables.

The tool supplies data types that generalize DBMS specific data types. Each column in a database schema is associated with one of the data types, e.g., *FirstName* in table *Employee* uses the *varchar* data type. The user can change the default data type and specify additional data-domain constraints. The tool supplies 39 data types, including all simple SQL:2003 data types. In addition, a number of real-life data types such as first name, last name, and zip-code are supplied. Advanced data types such as moving-object simulations and images are also supplied. The tool translates DBMS specific data types to the supplied data types, e.g., both the Oracle *VARCHAR2* type and the SQL Server *VARCHAR* type are mapped to the supplied *varchar* data type.

2.2 Column and Row Dependencies

This section explains the different dependencies possible in database schemas and how these are handled by the tool. Four different types of dependencies are possible. The first is a foreign-key constraint that is modeled as an edge in the graph. The second is an intra-row dependency, e.g., the dependency between the *StartDate* and *EndDate* columns of the *Works_On* table. The *StartDate* must always be smaller than the *EndDate*. This is achieved by basing the value of *EndDate* on the value of *StartDate*. To do this the user *binds* the *EndDate* column to the *StartDate* column. This binding gives the data type for the *EndDate* column access to the last generated value for *StartDate*. The intra-row binding between columns must form an acyclic graph. The third possibility is an intra-column dependency, i.e., the value of a column dependent on earlier generated values. As an example, the temperature in a refrigerator that dependents on the last 10 generated temperatures is implemented by storing the last 10 values in a data type. The second and third type of dependencies can be combined, e.g., 5 different refrigerators dependent on their last 10 values. This is implemented using a dictionary with the refrigerator id as the key that points to the previously generated values. The fourth possibility is intra-table dependencies between columns not specified as foreign keys, e.g., redundant data stored to optimize a query. The tool does not support this type of dependency.

3. DATA GENERATION

This section explains the main principles of the data generation algorithm. For details, please see [2]. It is assumed that the schema to generate data for is empty, i.e., there are no rows in any table.

3.1 Adjusting the Graph Model

The data will be simple if the database schema is empty and no adjustments are made, e.g., all primary key values are referenced once and the column data is build using default settings. The user can adjust all of these variables. This is shown here by using the schema in Figure 1. The participation is set to 100% between the *Works_On* and *Project* tables, to 80% between the *Works_On* and *Employee* tables, to 10% between the *Employee* to *Employee* table, and to 100% between the *Project_Detail* and *Project* tables. The distribution of how many times each of the primary key values are referenced is also adjusted. To model that each of the projects has between 6 and 14 employees associated, a normal distribution is chosen for the edge between the *Works_On* and *Employee* tables with a mean of 10, a standard deviation of 3, a minimum of 6, and a maximum of 14. To further improve the realism of the generated data, the data type is changed for some of the columns. The data type for the *FirstName* and *LastName* columns in the *Employee* node is changed to the specialized *firstname* and *lastname* data types. The specialized *firstname* data type has an option that allows the user to set the percentage of male and female names. Since the example models a software development company, the male percentage is set to 70%. The *EndDate* column on the *Works_On* table is bound to the *StartDate* column. The *date-time* data type provides options for specifying a distribution that models the time to add to the column to which it is bound (can be negative).

The final adjustment assigns a *function* to the *ExpectedPrice* column of the *Project_Details table*. The Excel-like function creates a dependency between the *ExpectedPrice* and *ExpectedManHours* columns. The function is declared as *Mult(*ExpectedManHours*;250)*.

3.2 Generation Algorithm

The main principle of the data generation is a depth-first traversal of the graph. The challenge in the algorithm is to generate and exchange composite and overlapping primary, unique, and foreign keys. The traversal begins at non-referenced nodes, i.e., nodes not referenced by other nodes. This is the *Works_On* node in Figure 3. If no such node exists the schema has cycles and random edges are altered to break these cycles. The next step is then to examine all out-bound edges of the chosen start node.

Three different types of edges can be encountered. A *Normal* edge means that values are needed for a foreign key in the referenced node (table), but that data for this node has not yet been generated. A *Forward* edge means that the data for the referenced node has already been created and foreign-key data is available. A *Backward* edge indicates a cycle. This means that the referenced node needs data (directly or indirectly) from the current node. Temporary values are then used in the current node for the foreign key. These temporary values are later replaced with real values when the data generation has completed for all other nodes. Two out-bound *Normal* edges can be traversed from the *Works_On* node in Figure 3. Assume that the edge pointing at the *Employee* node is traversed first. In the *Employee* node only one out-bound *Backward* edge exists (a cycle). The cycle is simple, but the technique used for dealing with *Backward* edges applies equally well to more complicated cycles. Data is generated for the *Employee* node since no further traversal is possible. The data is generated one row at a time. Data for the columns are generated in an order based on a topological sort of the bound columns and functions. None of the columns in *Employee* are bound or has functions assigned, so the order of columns is based on their ordinal position. 70% male names are generated for the *FirstName* column, and temporary values are generated for the *Works_For_FK* column (a cyclic foreign-key). When the generation for the *Employee* node has ended the algorithm returns to the *Works_On* node and the edge to *Project* is traversed. This continues until data for all tables has been created. When data is generated for all nodes what remains is to exchange the temporary values for the cyclic foreign-key in the *Employee* node with real values.

3.3 Non-Empty Database Schemas

One-click data generation is possible using both empty and non-empty database schemas. When using a non-empty schema, statistical information is automatically collected from the existing data and used to increase the realism of the generated data. The information includes cardinalities and participations of foreign keys. A normal distribution is used as default but this can be changed by the user. If changed, the new distribution is re-calculated based on the source data. Each column is automatically associated with a data type. The data type can collect statistics about the data in the original column. This statistical data is passed to the data type as a forward-only stream. The stream can be ignored, e.g., the *firstname* data type uses a local database with names and no statistics are needed. Other data types such as *integer* and *varchar* uses the source data to create descriptive distributions. The use of distributions limits the memory footprint. If a column's data type is changed by the user, the new data type also has access to the forward-only stream. To generate null values, where appropriate, the percentage of null values in the source data is stored with each column. The user can adjust this percentage.

4. PERFORMANCE

Figure 4 shows the performance of the tool, implemented in C#, when generating data for the TPC-C benchmark [1] (AMD Sempron 1,8 GHz, 1GB RAM). The tool shows linear scale-up and generates approximately 1.7MB data per second. The tool has been tested up to 400 GB of data on various OLTP and OLAP schemas. It shows linear scale-up even for the large data sets.

The general tool is approximately 2 times faster than a dedicated Perl program, but approximately 5 times slower than a dedicated C program. The speed of the data generation is dependent on the structure of the database schema. On the

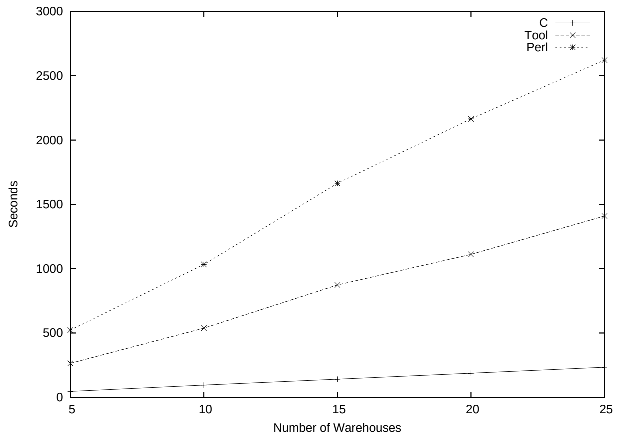

Figure 4: TPC-C Performance

schema for the Wisconsin benchmark the tool creates 2.6 MB/s. Figure 5 shows the performance when generating

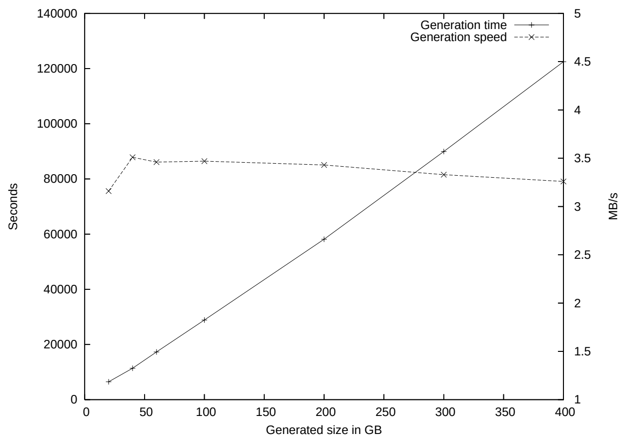

Figure 5: Data-Warehouse Performance

data for a real-world data warehouse (1 fact table, 5 dimension tables, and 8 outbreak or bridge tables [9]). The x-axis is the size of the generated data in GB. The left y-axis is the total number of seconds used for the generation. The right y-axis is the speed of the generation in MB pr. second.

In this test, the dimension tables are set to a fixed size and the fact table is scaled to achieve the required data sizes. The results show almost linear scale-up for data sizes up to 400 GB. The speed of the generation is approximately 3.2 MB/s and it is independent of the size of data generated. The small decrease in the generation speed is due to increased time used on sorting keys while randomizing. Note that it is considerablely faster to generate data for this data-warehouse schema (3.2 MB/s) than for the TPC-C schema (1.7 MB/s).

In general, the tool is typically very fast when generating data for a star schema because the large fact table is created by combining already generated keys from the dimension tables. These keys can typically be held in memory.

5. RELATED WORK

A recent article [4] presents a language approach for generating synthetic data. The main purpose of this language is to allow for generation of data that conforms to exact characteristics such as a normal or a Zipfian distribution. The language approach is very good at concisely specifying how to generate synthetic data. The graph-model approach presented in this paper is advantageous when the schema to generate data for is large and when existing data can be used as an outset for the data generation. With respect to large schemas the language approach has no automatic handling of foreign-key constraints. This is left entirely to the user. In the graph-model approach the model is build automatically from metadata and can be adjusted by the user. Note that it will require fundamental changes to the language approach to do this in a similar automated fashion. With respect to existing data it is again in the language approach left entirely to the user to specify the distribution of data for each column in the schema. In the approach taken in this paper the tool suggests distribution based on existing data that the user can adjust.

Freely available test data generation tools exist for most database benchmarks, e.g., tools for most TPC benchmarks can be downloaded from [1]. These tools are however very specialized, in contrast to the general tool presented here. [7] describes how to generate a billion records in less then 30 minutes using a cluster of high-end computers. An approach for generating consistent test data based on first order logic is presented in [8]. A part of [5] describes a data generation tool included in a framework for testing database applications. [3] uses a tree structure to automatically generate synthetic XML data according to a number of inputs. The system supports arbitrary complex structures, and can contain recursive definitions.

6. REFERENCES

[1] TPC homepage. www.tpc.org, Jun. 2006.

[2] www.data-generation.com, Jun. 2006.

[3] A. Aboulnaga, J. F. Naughton, and C. Zhang. Generating synthetic complex-structured XML data. In *WebDB*, pages 79–84, 2001.

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

[5] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber. A framework for testing database applications. In *ISSTA '00*, pages 147–157, 2000.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* The MIT Press, 2 edition, 2001.

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

[8] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data: Restricting the search space by a generator formula. *The VLDB Journal*, 2(2):173–214, 1993.

[9] European Internet Accessibility Observatory home page. www.eiao.net, Jun. 2006.