A Parallel General-Purpose Synthetic Data Generator

Joseph E. Hoag, Craig W. Thompson
Computer Science Computer Engineering Department
University of Arkansas
{jhoag, cwt}@uark.edu

Abstract

PSDG is a parallel synthetic data generator designed to generate "industrial sized" data sets quickly using cluster computing. PSDG depends on SDDL, a synthetic data description language that provides flexibility in the types of data we can generate.

Introduction

The IT industry need synthetic data generation tools for applications including:

- *Regression testing.* Repeatedly generate the same large data set for testing enterprise applications.
- *Secure application development.* Develop enterprise applications using generated data that is realistic but not real.
- *Testing of data mining applications.* Generate data sets with known characteristics to gauge whether data mining tools can discover those characteristics.

Several synthetic data generation tools already exist in the commercial world [1-4]. These tools can generate modest amounts of easily described data. However, they do not scale for generating "industrial-sized" (i.e., terabyte) data sets.

Data generation tools have recently been developed in the academic world as well [5-7]. These present new concepts in the form of graph- and language-oriented synthetic data description, providing greater flexibility in the description and generation of synthetic data.

In this paper, we present the Parallel Synthetic Data Generator (PSDG). PSDG is designed to generate data across multiple processors. This allows PSDG to harness the power of cluster/grid computing to generate huge data sets with linear speedup. Parallel execution involves more than launching several generators simultaneously. PSDG will generate constant output (for a given input file) regardless of the degree of parallelism. This repeatable determinism is a requirement for regression testing applications.

PSDG is written in Java so it is portable across platforms. Although PSDG supports direct-to-database data generation, it is generally quicker to generate the data to file(s) and then use a fast load utility to upload data into a database.

This paper also describes Synthetic Data Description Language (SDDL). SDDL files serve as input to PSDG. SDDL is an XML-based language that codifies the manner in which generated data can be described and constrained. Rich description mechanisms available in SDDL enable generation of a variety of kinds of synthetic data.

In the following section, we document features of SDDL. Next, we discuss the concept of pools. Then we describe mechanisms that produce deterministic data sets over an arbitrary number of processors. We conclude with performance data and future directions.

SDDL

SDDL is an XML-based language that provides the user flexibility in describing and constraining synthetically generated data. While SDDL was developed along with PSDG, it is intended to be expressive and could be used as input by other synthetic data generators. SDDL is similar in concept to DGL [5] with important differences:

- SDDL is XML-based while DGL is C-like
- SDDL constructs are designed to produce identical results across any number of generation processes
- While DGL provides powerful stream-based data generators, SDDL generation constructs are more modular and understandable

SDDL describes at its outer level a "database" element, which is composed of zero or more pool elements and one or more table elements. Pools will be described later. The following is a valid (if trivial) SDDL file:

```
<?xml version="1.0" encoding="UTF-8"?>
<database>
 <seed>1240958412</seed>
 <pool name="colors">
  <choice="red"/>
  <choice="green"/>
  <choice="blue"/>
 </pool>
 <table name="PAIRS" length="5">
  <field name="F1" type="int">
   <min>1</min>
   <max>50</max>
  </field>
  <field name="F2" type="CHAR(5)">
```

```
    <formula>colors</formula>
   </field>
  </table>
</database>
```

The file above, when input to PSDG, would result in the synthesis of a relation named **PAIRS**, which might look like:

F1	F2
34	green
10	blue
47	green
17	red
8	red

A table element contains all information needed to synthesize a table, including a name attribute, a length attribute, and one or more embedded field elements. The attributes associated with the generated table are defined by the field elements within the table element.

Each field element has "name" and "type" attributes. Field types can be one of **int**, **real**, **string**, **bool**, **date**, **time** or **timestamp**. Field types can also be defined as standard SQL data types, i.e., **CHAR(n)**, **VARCHAR(n)**, **NUMERIC(m,n)**.

Each field also contains a generation constraint. SDDL currently supports five types of generation constraints: min/max, distribution, formula, iteration, and queryPool. Each are described below.

Min/Max Constraints

The user can specifies a minimum and maximum value for a field, e.g.,

```
<field name="age" type="int">
 <min>20</min>
 <max>65</max>
</field>
```

In this example, generated values for the **age** attribute are randomly distributed between 20 and 65, inclusive.

Distribution Constraints

A distribution constraint is a set of min/max constraints (called "tiers"), each of which has a specified statistical probability of being selected:

```
<field name="age" type="int">
 <dist>
  <tier prob="0.50" min="20" max="30"/>
  <tier prob="0.30" min="31" max="50"/>
  <tier prob="0.20" min="51" max="65"/>
 </dist>
</field>
```

In this example, the generated **age** column values would have a 50% chance of being between 20 and 30, a 30% chance of being between 31 and 50, and a 20% chance of being between 51 and 65.

Formula Constraints

Using a formula constraint, a field can be defined in terms of a mathematical formula consisting of operators, constants, built-in functions, and other field values, for example:

```
<field name="ship_date" type="date">
 <formula>order_date+IRND(4)</formula>
</field>
```

In this example, the **ship_date** field will be equal to the **order_date** field plus anywhere from 0 to 3 days.

Formula constraints can also contain pool references as discussed in the Pools section below.

Iterations

Iterations constrain the generator to iterate through a set of values for a specified column. If a table contains one or more iteration-constrained fields, then the length of the table will be governed by the iteration results. SDDL supports three kinds of iterations: query, pool, and count iterations.

Query iterations allow a column to iterate through the results of a query. For example:

```
<field name="store_nbr" type="int">
 <iteration query="select store_nbr from stores"/>
</field>
```

In this example, the generated **store_nbr** column values would iterate through the **store_nbr** column values in the **stores** table.

Similarly, iterations can iterate through pool choices:

```
<iteration pool="states"/>
```

or numeric values:

```
<iteration base="1" count="100"/>
```

By default, an iteration results in one row of output for each element of its set. However, the user can specify repetition of iteration elements:

```
<field name="F1" type="int">
 <iteration base="1" count="5">
  <repeatMin>3</repeatMin>
  <repeatMax>6</repeatMax>
 </iteration>
</field>
```

In the example above, the **F1** attribute would contain the values 1 through 5, each repeated anywhere from 3

to 6 times. Iteration repeat constraints can also be specified with a statistical distribution:

```
<field name="F1" type="int">
 <iteration base="1" count="5">
  <repeatDist>
   <tier prob="0.70" min="3" max="5"/>
   <tier prob="0.30" min="6" max="6"/>
  </repeatDist>
 </iteration>
</field>
```

If there are multiple iteration-constrained fields in a table, they are considered to be nested, with the nesting order the same as the order of appearance in the table description. Consider the following table definition:

```
<table name="nesting_example">
 <field name="A" type="int">
  <iteration base="5" count="3"/>
 </field>
 <field name="B" type="int">
  <iteration base="100" count="2"/>
 </field>
</table>
```

The **nesting_example** table will *always* generate:

A	B
5	100
5	101
6	100
6	101
7	100
7	101

Query Pools

Query pools are used to enforce referential integrity constraints. Using query pools, the value of a field is chosen from the result of a query. In the following example, values generated in the **StudentID** field would be valid IDs from the **students** table.

```
<field name="StudentID" type="int">
 <queryPool>select ID from students</queryPool>
</field>
```

Pools

Pools are hierarchically structured SDDL elements that can serve as user-defined domains, sources of reference information, and modeling tools.

In SDDL, a pool element must have a **name** attribute and is composed of one or more choice elements. Each choice element must have a **name** attribute and can optionally be composed of sub-pools and auxiliary data items. Pools are modular. Reusable pools can be stored in separate files and imported into SDDL files.

The following pool provides a domain for **states**:

```
<pool name="states">
 <choice name="AK"/>
 <choice name="AL"/>
 <choice name="AR"/>
 ...
 <choice name="WV"/>
 <choice name="WY"/>
</pool>
```

The pool can be accessed from a formula constraint as follows:

```
<field name="state" type="CHAR(2)">
 <formula>states</formula>
</field>
```

Using the formula, each entry in the "states" pool has a 1-in-50 chance of being output for each row of the "state" field. However, we could make the "states" pool more interesting:

```
<pool name="stateZip">
 <choice name="AK">
  <pool name="zips">
   <choice name="99501">
    <city>ANCHORAGE</city>
    <county>ANCHORAGE</county>
    <weight>16211</weight>
   </choice>
   <choice name="99502">
    <city>ANCHORAGE</city>
    <county>ANCHORAGE</county>
    <weight>18626</weight>
   </choice>
   ...
  </pool> <!—end of "zips" sub-pool-->
  <weight>624992</weight>
 </choice> <!—end of "AK" choice-->
 ...
 <choice name="WY">
  <pool name="zips">
   <choice name="82001">
    <city>CHEYENNE</city>
    <county>LARAMIE</county>
    <weight>34767</weight>
   </choice>
   <choice name="82007">
    <city>CHEYENNE</city>
    <county>LARAMIE</county>
    <weight>15840</weight>
   </choice>
   ...
  </pool> <!—end of "zips" sub-pool-->
  <weight>493502</weight>
 </choice> <!—end of "WY" choice-->
</pool> <!—end of "stateZip" pool-->
```

The **stateZip** pool above is a population model of the U.S. taken from publicly available *2000 Census* data (www.census.gov). At the top level, there are still 50 states in the pool. Each state contains a pool of zip codes; each choice in the **zips** sub-pool contains the city and county associated with the zip code. Both the top-level **states** pool and the nested **zips** pools are weighted by population.

Consider the following SDDL table definition that uses the **stateZip** pool:

```
<table name="offices">
 <field name="OfficeID" type="int">
  <iteration base="1000" count="1000"/>
 </field>
 <field name="state" type="CHAR(2)">
  <formula>stateZip</formula>
 </field>
 <field name="zip" type="CHAR(5)">
  <formula>stateZip[state].zips</formula>
 </field>
 <field name="city" type="CHAR(25)">
  <formula>stateZip[state].zips[zip].city</formula>
 </field>
</table>
```

The **offices** table generated from the definition above will have information for 1000 offices. Offices will be distributed within the U.S. according to inter-state and intra-state population statistics. The **state**, **city**, and **zip** fields generated for each row will be consistent with each other.

Given this structural and semantic flexibility, pools can be used to describe a variety of complex data types. Using pools, we have so far modeled such diverse concepts as graphs, maps, state machines, and context-free grammars.

Parallel Data Generation

Multi-processor systems (clusters, grids) are increasingly affordable. The size of data sets is growing – terabyte-sized tables are not uncommon. Why not take advantage of the former to synthetically generate the latter?

We designed PSDG with parallel generation capability as a requirement. We wanted to minimize communication between generation processes and maintain deterministic output (for a given input) regardless of the degree of parallelism.

Previous synthetic data generation frameworks have supported parallelism to some degree. Gray et al. [8] described methods for writing special-purpose data generators in parallel. The MUDD generator [11] provided a general purpose (though simple) input language, and decoupled the data description from the parallelization details. The KRDataGenerator [12] (a commercialization of [6]), which is graph-based, has a distributed generation capability but can not describe data with the detail of SDDL.

PSDG distinguishes itself by providing a very descriptive input language (SDDL) while supporting easy parallelism. Like MUDD, PSDG decouples data generation details from data description; users need not take parallelism details into account when constructing an SDDL file.

Each PSDG generation process is launched with the knowledge of how many processes are participating, as well as its own process index. With this information, a generation process can determine the extent of the data that it is responsible for generating without the need for inter-process communication.

PSDG slices the generated data horizontally between generation processes. Generation processes handle slices in a "striped" fashion: process 0 of N will generate slices {0, N, 2N,…}, process 1 of N will generate slices {1, N+1, 2N+1,…}, and so on. We can use two separate methods of data slicing: Algorithm 1 is used when there are no iterations in the SDDL description (for a table), and Algorithm 2 is used when iterations are present.

Generation Algorithm 1

With no iterations present, a table's row structure is predictable, and its rows are divided into "swaths", each of size **SWATHSIZE**. The generation processes generate swaths in an alternating round-robin fashion. Before generating a swath, the generation process will call the re-seed function **RF(seed, row)** (where **seed** is the user-specified seed, and **row** is the start row of the swath), which uses **seed** and **row** to re-seed the random number generator.

If there is only one generation process, generation proceeds as follows (assuming **SWATHSIZE = 100**):

Process 0:
***RF*(*seed*,0)**
Generate rows 0-99
***RF*(*seed*,100)**
Generate rows 100-199
***RF*(*seed*,200)**
Generate rows 200-299
...

If there are two generation processes, the generation proceeds in parallel as follows:

Process 0:	Process 1:
RF(*seed*,0)	*RF*(*seed*,100)
Generate rows 0-99	Generate rows 100-199
RF(*seed*,200)	*RF*(*seed*,300)
Generate rows 200-299	Generate rows 300-399
…	…

The random number generator is always re-seeded to a deterministic value (based on the user-specified seed and the row number) before generating a swath. Thus any particular swath will be generated identically regardless of the number of processes participating in the generation.

Note that the load balancing for Algorithm 1 is even; each generation process generates at most **SWATHSIZE** more rows than any other generation process.

Generation Algorithm 2

When iterations are present, it becomes more difficult to deal out constant-sized swaths to each generation process. Consider the following SDDL snippet:

```
<table name="iteration_example">
  <field name="deptID" type="int">
    <iteration query="select ID from departments"/>
  </field>
  <field name="courseID" type="int">
    <iteration query="select ID from courses where
        deptID = [deptID]"/>
  </field>
</table>
```

How many **courseID** rows will be generated for each **deptID** value? We can't really tell before running the actual queries. As a consequence, we don't know *a priori* how to divide the output into equal-sized slices.

However, we do know the number of elements in any outer iteration element. For query iterations, it is the number of elements returned by the given query. For pool iterations, it is the number of choices in the specified pool. For numeric iterations, it is the value of the **count** attribute. Therefore, the table is sliced up into outer iteration elements (OIEs).

When a single generation process performs generation algorithm 2, the following occurs:

Process 0:
RF(*seed*,0)
Generate OIE 0
RF(*seed*,1)
Generate OIE 1
RF(*seed*,2)
Generate OIE 2
…

With 2 processes, the generation proceeds as follows:

Process 0:	Process 1:
RF(*seed*,0)	*RF*(*seed*,1)
Generate OIE 0	Generate OIE 1
RF(*seed*,2)	*RF*(*seed*,3)
Generate OIE 2	Generate OIE 3
…	…

Again, the random number generator is always re-seeded to a deterministic value before generating the rows associated with an outer iteration element. Thus the rows associated with any OIE will be generated identically regardless of the number of processes participating in the generation.

Note that Algorithm 2 is balanced with respect to the number of OIEs assigned to the generation processes; each generation process will handle the generation of at most one more OIE than its peers. However, the number of rows associated with an OIE is not guaranteed to be constant. Therefore, it is possible for the generation to be unbalanced in terms of the number of rows generated per generation process.

Performance

As a reference point for generation speed, we generated the SetQuery [9] and TPC-C [10] benchmark data sets:

	Generated MB/Second	
Data Set	**1 processor**	**2 processors**
Set-Query	5.41	10.69
TPC-C (W=1)	3.58	6.73

The results above were obtained using Pentium 4 processors running at 3 GHz. The slightly sub-linear speedup is due to serial functionality embedded in our current iteration logic (which we are removing).

More importantly, we've benchmarked PSDG in real-world applications. We recently collaborated with a major retailer on a project involving the generation of 10 years worth of realistic store-item-sales data, resulting in 70 billion rows, or nearly 5 Terabytes of data. This data had fairly complex inter- and intra-row dependencies. Running PSDG across 16 1.6-GHz Itanium processors, we were able to reach data generation speeds of over 500,000 rows/second.

Conclusion and Future Directions

SDDL provides the functionality and expressiveness needed in a synthetic data description language. Pools provide for user-defined domains and data dictionaries. Formulas and pools allow for intra-row dependencies. Min/max and distribution constraints support loose inter-row dependencies, and iteration variables (not discussed here) support tight inter-row dependencies. Query pools and query iterations allow for inter-table

dependencies. SDDL pools allow for modeling a rich array of concepts, from simple domains and reference tables to graphs, maps, state machines and context-free grammars.

While SDDL does not currently support the complex statistical distributions found in [5] and [8], it does support the use of user-defined plugin functions in formulas. Complex data distributions (such as Gaussian and Zipfian distributions) can be enforced using this mechanism.

The data generator itself (PSDG) provides partitioning algorithms that allow for the parallel generation of data sets. This parallel capability makes it possible to generate large "industrial size" data sets quickly. Our partitioning algorithms balance processor loads and make sure that the data generated from a given input will be the same regardless of the number of processors across which it is generated. Such deterministic behavior makes PSDG useful for generating regression test data sets.

In our design of both SDDL and PSDG, we aimed to satisfy two goals: (1) provide a rich, flexible, extendible synthetic data description language, and (2) support deterministic parallel generation of data sets. When these two goals have conflicted, we have so far given priority to the latter. For example, allowing for "side-by-side" iterations, in addition to nested iterations, would enhance the descriptive power of SDDL. However, side-by-side iterations would be difficult to partition and parallelize, so we have so far chosen not to implement them.

A number of improvements to PSDG/SDDL are planned:

- Add SDDL constructs to allow PSDG to handle circular inter-table data dependencies and simultaneous multi-table generation
- Add support for passing environment and command-line generation variables to PSDG, allowing for the use of a single SDDL file to generate a number of different data sets
- Add streaming output for applications that require continuous feeds

In addition to its XML-based specification language (SDDL), the PSDG engine currently provides a GUI-based user interface so users can edit SDDL specifications and control the generation of data sets using a more user friendly interface. The PSDG engine also includes an ODBC connection to remote relational databases so data and pools can be imported or exported. It is often important to sanitize a real dataset before release to another organization. We are working on extending PSDG to extract statistical and descriptive data from relational databases to facilitate auto-generation of scrubbed data sets. The user will specify the desired degree of reality of the scrubbed copy that PSDG will generate.

In addition to working to improve our data generation framework, we are collaborating with industrial partners to extend the modeling capability of SDDL. One significant target is realistic supply chain data showing RFID reads throughout a supply chain that spans manufacturing, transport, warehousing, and retailing.

For more information on this project, see [13].

References

[1] Turbo Data, http://www.turbodata.ca

[2] GS Data Generator, http://www.GSApps.com

[3] DTM Data Generator, http://www.sqledit.com

[4] RowGen, http://www.iri.com

[5] N. Bruno and S. Chaudhuri. "Flexible Database Generators," *Proceedings of the 31st VLDB Conference*, pp.1097-1107, 2005.

[6] K. Houkjaer, K. Torp, and R. Wind. "Simple and Realistic Data Generation," *Proceedings on Very Large Data Bases*, 2006, pp. 1243-1246.

[7] P. Lin et al. "Development of a Synthetic Data Set Generator for Building and Testing Information Discovery Systems," *Proceedings of the Third International Conference on Information Technology: New Generations*, IEEE Computer Society, Las Vegas, USA, April 10-12, 2006, pp. 707-712.

[8] J. Gray et al. "Quickly Generating Billion-Record Synthetic Databases," *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1994.

[9] P. E. O'Neil. The Set-Query Benchmark. www.cs.umb.edu/~poneil/SetQBM.pdf

[10] Transaction Processing Performance Council, http://www.tpc.org/tpcc

[11] J. Stephens and M. Poess, "MUDD: a Multi-Dimensional Data Generator", *International Workshop on Software and Performance*, Redwood City, California, January 2004, pp. 104-109.

[12] KRDataGeneration home page, http://www.data-generation.com, accessed January 2007.

[13] University of Arkansas Synthetic Data Generation home page, http://www.csce.uark.edu/~cwt/SDG.