Rapid Development of Data Generators Using Meta Generators in PDGF

Tilmann Rabl[1], Meikel Poess[2], Manuel Danisch[3], Hans-Arno Jacobsen[1]
[1]Middleware Systems Research Group, University of Toronto
[2]Oracle Corporation, Redwood Shores, CA-94404
[3]Faculty of Computer Science and Mathematics, University of Passau

ABSTRACT

Generating data sets for the performance testing of database systems on a particular hardware configuration and application domain is a very time consuming and tedious process. It is time consuming, because of the large amount of data that needs to be generated and tedious, because new data generators might need to be developed or existing once adjusted. The difficulty in generating this data is amplified by constant advances in hardware and software that allow the testing of ever larger and more complicated systems. In this paper, we present an approach for rapidly developing customized data generators. Our approach, which is based on the Parallel Data Generator Framework (PDGF), deploys a new concept of so called meta generators. Meta generators extend the concept of column-based generators in PDGF. Deploying meta generators in PDGF significantly reduces the development effort of customized data generators, it facilitates their debugging and eases their maintenance.

Categories and Subject Descriptors

K.6.2 [**Management of Computing and Information Systems**]: Installation Management—*Benchmarks*

General Terms

Measurement, Performance

Keywords

PDGF; data generation; meta generators

1. INTRODUCTION

In benchmarking database management systems (DBMS) a time consuming and highly repetitive task is the development of data generators and the generation of test data. For the testing of commercial DBMS industry standard benchmarks may be used. However, new features, not anticipated by the developers of standard-benchmarks, often require specialized data to cover all aspects of the feature, especially to cover important border cases. This is amplified by constant advances in hardware that allow the deployment of ever larger and more complicated systems involving exabytes of data and thousands of computer systems. Before purchasing new DBMS, standard benchmarks may be used. However, in an increasing number of cases customers are demanding tests that use data more similar to theirs. Database administrators often face similar issues when provisioning systems. They need to scale their existing data sets to assure that their production systems can handle future load even during peak business times. As a consequence, there has been quite a lot of research conducted on data generation for DBMS testing. An important milestone was the paper by Gray et al. [7], the authors showed how to generate data sets with different distributions and dense unique sequences in linear time and in parallel.

Most existing data generators are special purpose implementations for a single data set, like those developed for industry standard or scientific benchmarks. Examples for special purpose data generators developed for industry standard benchmarks are, Dbgen for TPC-H [13], DsDgen (aka MUDD) for TPC-DS [20], Egen for TPC-E, and Workload Generator for Storage Performance Council Benchmarks [19]. Examples of special purpose scientific data generators are the generators for SetQuery [12], YCSB [4], and the Wisconsin database benchmark [2]. However, all of these data generators are special purpose implementations that can only generate one type of data set, usually in varying size.

In many cases existing special purpose data generators cannot be used for the testing of DBMS. They cannot be used when innovative features are tested, a new schema design is tested, arbitrary data scaling is needed, or when the data must follow a very specific distribution. Developing new data generators from scratch is often not an option because it is very time and resource intensive. Using general purpose data generators seems to be a feasible alternative. In [3] Bruno and Chaudhuri present a data generation framework that largely relies on scanning a given database to generate various distributions and interdependencies. Two other tools that offer similar capabilities are MUDD [20] and PSDG [9]. Both feature description languages for the definition of the data layout and advanced distributions. Another common approach are graph based models as presented by Houkjær et al. [10] and Lin et al. [11]. The active demand for these generic data generation tools also feeds a lively industry in this niche, such as Red Gate SQL Data Generator [18], DTM Data Generator [5], and GS DataGenerator [8].

The Parallel Data Generation Framework (PDGF), developed at the University of Passau, is a generic data generator. It was designed to take advantage of today's multi-core processors and large clusters of computers to generate exabytes of synthetic benchmark data. PDGF uses a fully computational approach and is a pure Java implementation which makes it very portable. Version 1.0 [15] was capable of generating data for complex schemas that contain inter

DBtest '13, June 24 2013, New York, NY, USA

Figure 1: Schematic overview of the value generation steps in PDGF

column and intra table dependencies [16], as used in TPC-H. It was enhanced with features to generate update data (Version 2.0) [6].

While configuring PDGF for complex data generators, such as those used in benchmarks of the Transaction Processing Performance Council (TPC), it became apparent that many columns share large portions of their data generation logic. For instance, consider the following NULL feature. Assuming each nullable field of a database table $T\{C_1, C_2, .., C_n\}$ ought to be set to NULL with a likelihood of $L\{P_{C1}, P_{C2}, ..., P_{Cn}\}$, then the code to generate data for each nullable column needs to implement a NULL feature. This results in the replication of the same or similar data generation logic in various places.

In this paper, we introduce the concept of *meta field value generators* (meta-FVG), an extension to the concept of field value generators (FVG) of PDGF Version 1. FVGs are used to generate data for table columns. meta-VFGs can be used to implement common functionality across FVGs, to combine values of multiple FVGs, and to implement logical dependencies between FVGs. The use of meta-FVG dramatically reduces the number of FVGs needed to configure PDGF for a particular data generator, thereby considerably increasing the speed at which data generators can be created and significantly easing the maintenance of the code base. Our main contributions are: i) We present the principle of meta field value generators (meta-FVG), a methodology that reduces the complexity of implementing field value generators and increases the flexibility of schema composition, while greatly increasing programmer's productivity, ii) We identify a set of core FVGs and meta-FVGs that allows for a wide range of data generation including TPC-H, iii) We demonstrate the benefit of meta-FVGs in three use cases, namely the PDGF TPC-H implementation [14], the PDGF SSB implementation [17] and an data generator for an ETL benchmark.

The remainder of this paper is organized as follows. In Section 2, we give a brief overview of the Parallel Data Generation Framework. Section 3 gives a formal introduction to the concepts of field value generators and meta field value generators. Section 4 briefly recaps how FVG are defined in PDGF and which FVG are built into the core of PDGF. Section 5 puts the concept of meta field value generators in context of PDGF. Section 6 lists two use cases which demonstrate the benefits of using meta-FVGs. Finally, Section 8 concludes with future work.

2. PARALLEL DATA GENERATION FRAMEWORK

In this section we, will give a brief introduction to PDGF. Detailed descriptions of the architecture and functionality can be found in previous publications [15, 6]. PDGF is a generic data generation framework that was built around the principle of parallel pseudo random number generation. PDGF exploits the inherent parallelism of xorshift-based random number generators to generate every single field value of a table independently. The random number generator in PDGF works like a hash function. Therefore, it is possible to generate any number in a random number stream without calculating the complete stream. In order to achieve the same independence in the value generation, it is constrained to deterministically repeatable calculations. The abstract process of data generation in PDGF can be seen in Figure 1. Starting from the ID of a value – e.g. the row number – a random number is generated. The random number generator (RNG) is seeded in a way that it generates a separate stream for every column in a database. The random number is passed to a field value generator (FVG), which calls the random number generator for a random number and uses it to generate deterministically a field value. A frequently used FVG is a dictionary look up FVG. It is used to generate names, streets, colors, etc. The generated value is written to a file by the output module. All of these components are exchangeable and extendable by plugins in PDGF. Nevertheless, PDGF has a set of versatile RNGs, a set of most commonly used FVGs and flexible output modules built in. We refer to them as *core* components.

PDGF is easily configurable by using two configuration files, the schema config and the generation config. As explained above, the actual data for each column of a database table is generated in FVGs. FVGs can be parametrized. The mapping of database columns to FVGs is specified in the schema config. PDGF contains a set of core FVGs that cover commonly used value generations, such as number, string and date. We will give more details on the core FVGs in Section 4. To generate custom data, PDGF allows for custom written FVGs in Java that can be plugged in via an API. This makes PDGF easy to use, yet very powerful. The generation configuration file describes the output and scheduling. That allows for a partial generation or different kinds of output formats of the same logical data. PDGF combined with a specific set of configuration files and optional plug-in FVGs is referred to as a specific *data generator*. During the implementation of various data generators, we found that for a certain data set we needed additional properties for a certain FVG such as the optional generation of NULL values. Some of these properties, like number formatting, are necessary for multiple FVGs but applicable only for a subset of our core FVGs and sometimes only applicable for certain configurations of the FVGs, e.g. a numeric date format. Instead of implementing such generic properties in multiple FVGs, we introduced the principle of meta field value generators to encapsulate generic properties. In the next section, we will give a formal description of the idea of meta-FVGs.

3. FORMAL DESCRIPTION OF VALUE GENERATION

Figure 1 shows how values are generated for each field of a database table. Each field is given an identification (*id*). The *id* is calculated by the framework from the row number. This *id* is converted into a random number (*rn*) by the random number generator (*rng*), which is then used by the column's field value generator to generate a field value, e.g. number, string, or date. The random number generator is a function:

$$rng(id) = rn \tag{1}$$

More specifically, the parallel random number generator (prng) is seeded to generate a specific sequence for each series of values (i.e. a column in a table):

$$rng(ID) = prng(ID + seed) = rn \tag{2}$$

The seed is provided by the framework. PDGF's internal seeding strategy ensures a deterministic seed for every series of values (cf. [6]). The resulting random number is then passed to the field value generator. The FVG in turn is also a function that maps the random

number to a field value. The most simple FVG is a constant function that assigns the same value to each random number. Below is an example for a number FVG that generates a number between 0 and 100, e.g. for a person's age field:

$$ageGen(rn) = rn \,\%\, 101 \tag{3}$$

where $\%$ denotes the modulo operation. Due to its limited reusability, this $ageGen$ is not a separate FVG in PDGF instead there is a more generic number FVG where the range constraint (r) and offset (o) are parameters to the function:

$$numberGen(rn, r, o) = rn \,\%\, r + o \tag{4}$$

For some columns it might be desirable to generate *NULL* values with a specific probability p. This can be implemented using a conditional statement:

$$numNull(rn, r, o, p) = \begin{cases} rn \,\%\, \frac{r}{1-p} + o, & \text{if } rn \,\%\, \frac{r}{1-p} < r \\ NULL, & \text{else} \end{cases} \tag{5}$$

The definition of the $numNull$ generating function in Equation 5 increases the range of the number generator by the probability of *NULL* values and assigns *NULL* whenever the value is out of the original range. However, defining the NULL logic in each FVG is impractical. Therefore, we make use of the principle of higher order functions. Consider the following higher order function which makes a case distinction based on probability p (in percent):

$$case(rn, p, f, g) = \begin{cases} f(rn), & \text{if } rn \,\%\, 100 < p \\ g(rn), & \text{else} \end{cases} \tag{6}$$

Using this *case* function, we can build the $numNull$ function from the $case$ and $numberGen$ functions and a $nullGen$ function without changing their original definition. The $nullGen$ function generates only *NULL* values. In this context, we call $numberGen$ and $nullGen$ sub-FVGs of $case$. We call these higher order functions meta field value generators. This approach makes it possible to build many different complex FVGs with a small set of FVGs and meta-FVGs. An obvious problem that remains are the range requirements of the various FVGs. Prerequisite for every FVG is that the all possible random numbers can occur with the same probability. Therefore, we change the input for the FVG and meta-FVG to random number generators instead of random numbers. This way we do not have to worry about correlations in the ranges. A meta-FVG simply uses as many random numbers as it needs and passes a correctly seeded random number generator to the subsequent FVGs. To further increase the genericness of the approach it is also possible to have meta-FVGs as sub-FVGs.

4. FIELD VALUE GENERATORS IN PDGF

To clarify how all of this is put together in PDGF, in the following example we want to create a data generator to populate a simple table that contains user information. The user table has two fields, one for holding the name of the user and another for her/his age.

4.1 Schema Configuration Files

The *schema* file (schema.xml) of a specific instantiation of PDGF, i.e. a data generator, defines the basic structure and content of all data to be generated by it. Following the relational model, data is specified in terms of tables and columns. Each table $T\{C_1, .., C_n\}$ of a schema S is defined by a *table element*. Each table element includes at least one *size element* and one *fields element*. The fields element contains at least one *field element*, one for each table column $\{C_1, C_2, .., C_n\}$. The definition of each field element must include the specification of a data type and the assignment of an FVG that is used to generate its data. The assignment of FVGs to columns is specified in the *name attribute* of the *generator elements*. Column data types are specified with the *type element* within the *field element*.

```
<table name="users">
  <size>10000</size>
  <fields>
    <field name="name">
      <type>java.sql.types.VARCHAR</type>
      <size>100</size>
      <gen_DictList>
        <file>dicts/names.dict</file>
      </gen_DictList>
    </field>
    <field name="age">
      <type>java.sql.types.NUMERIC</type>
      <gen_LongGenerator>
        <min>0</min>
        <max>120</max>
      </gen_LongGenerator>
    </field>
  </fields>
</table>
```

Figure 2: Partial schema.xml for a minimal user table

4.2 XML Element Parsing

Another important topic is the parsing of the XML elements. In general, everything down to and including the generator element is parsed by PDGF itself. Everything within the generator element is parsed by the corresponding generator. The element parsing itself is done by node parsers that are implemented as internal classes. PDGF has common node parsers for parsing XML. FVGs can reuse most of these node parsers, but they can also implement their own. Every FVG must configure which node parsers it wants to use and if they are mandatory or optional, because PDGF checks the XML when the config file is loaded. If it finds an unknown element (not configured in the corresponding FVG) or if a required element of a FVG is missing, it rejects the config file and issues a comprehensive error message indicating why the config file was rejected.

4.3 Example: User Table

Figure 2 shows an excerpt of the minimal schema file to generate data for the user table. For a complete loadable config file, some project specific xml elements like the project seed or the default RNG must be added outside of the shown table element. The first tag *<table>* defines the table name to be "users". It is followed by a second tag *<size>* that defines the cardinality of the table to be 10,000. The following two sections define two columns, named *name* and *age*.

"name" is defined as a VARCHAR data type with a maximum field width of 100 characters. The values for name are generated using the *DictList* FVG. The DictList FVG picks random elements from a list of strings. The list of strings is configurable and presented to DictList as a file name. We use the *<file>* tag to present the file name to DictList, which is dicts/names.dict in our example. If there was an optional disableRng-element, DictList used the current line number for getting a value from the dictionary. Without the disableRng-element, the DictList FVG chooses a random line from the dictionary for every row of the table.

"age" is defined as a NUMERIC data type. It is assigned the *LongGenerator* FVG for generating field values between 0 and

```
<field name="age">
  <type>java.sql.types.NUMERIC</type>
  <gen_NullGenerator>
    <probability>0.05</probability>
    <gen_LongGenerator>
      <min>0</min>
      <max>120</max>
    </gen_LongGenerator>
  </gen_NullGenerator>
</field>
```

Figure 3: Definition of field age with a probability of 5% for NULL values

120. LongGenerator must be configured with a *min* and a *max* element, limiting the lower and upper bound of the generated numeric value. It can be configured with an optional distribution-element for changing the distribution of the random numbers. The default distribution is uniform.

4.4 Limitations of Field Value Generators

FVGs are a handy way to define the contents of table columns. Since they are defined separately from the columns themselves and since they can be parametrized, the same FVG can be used for the generation of multiple columns. Despite their re-usability, very large schemas with many similar but not equal columns and complex relationships require the implementation of many FVGs. For instance, consider NULL values of relational theory. A naive approach to injecting NULLs into the data for a column is to simply change its FVG to generate NULL values with a certain likelihood. However, this might conflict with other not nullable columns that use the same FVG. Another approach is to create a copy of the FVG and extend it to generate NULL values. However, this approach has some disadvantages: if a schema contains many nullable columns, many FVG need to be copied, renamed and extended to generate NULL values leading to code duplication and a maintenance nightmare. If each columns requires a different likelihood of NULL values, each FVGs needs to define its own method of NULL injection with its own parameter. These additional parameters may lead to a blown up schema configuration file. This problem worsens, when additional features are added to the FVGs, because the number of parameters each generator defines increases. We found that there should be a way of extracting common logic and functionality out of FVGs to a higher level, a meta level. This led us to the introduction of meta field value generators in PDGF.

5. META FIELD VALUE GENERATORS

A meta-field value generator is a generator that extracts common logic of multiple FVGs. Formally, meta-FVG can be considered as a higher-order function, i.e. a function that takes other functions as parameters (see Section 3). In PDGF, a meta-FVG takes FVGs and parameters as arguments.

5.1 User Table With NULL Values

Consider the example in Figure 2. Assume that in 5% of all rows the "age" field should contain a NULL value. This is analog to the examples in Section 3. Using only the concept of FVGs requires copying the LongGenerator to a NullLongGenerator and introducing a new parameter "probability" along with the corresponding code to manage the NULL generation. With the new meta-FVG concept introducing NULL values can be done without changing the LongGenerator. First we define a meta-FVG, e.g. NullGenerator, with the LongGenerator as generator subnode. This way the NULL generation code is encapsulated in the NullGenerator, while the code for simply generating numeric values is encapsulated in the LongGenerator. The field definition of "age" has to be altered to the XML code shown in figure 3. With the altered field, about 500 lines of "users" have NULL values in the "age" column, while the other lines have a number between 0 and 120.

In contrast to the age field definition in figure 2, the new one in figure 3 defines the NullGenerator as primary FVG. The NullGenerator is no FVG but a meta-FVG, however, this makes no difference to PDGF. Whenever a FVG is expected, a meta-FVG can be used as well. The NullGenerator has a mandatory probability-element which configures the likelihood of generating NULL as value. If it decides to not generate NULL, it uses the FVG defined in its generator element to generate a value for this field. The definition of the inner FVG is the same as in Figure 2. So we achieved our goal: the NullGenerator generates NULL with a likelihood of 5% and numeric values between 0 and 10 in 95% of all lines.

5.2 Meta Field Value Generators in PDGF

There are several meta-FVGs built into PDGF. In general, these core meta-FVGs can be split into two groups: *post-processing* meta-FVGs and *flow control* meta-FVGs. A post-processing meta-FVG uses its assigned FVG(s) to generate a basic field value and post processes this value in a defined way. It does not choose which FVG(s) to run, it simply executes the FVG(s) and post processes the generated data. A flow control meta-FVG does not alter the data generated by its FVG(s), but chooses which FVG to run. So flow control meta-FVGs contain pure structural decision logic, while post processing meta-FVGs contain strict data alteration logic. Although a meta-FVG could mix post-processing and flow control code, we decided to differentiate between those two types, because doing so greatly enhances the re-usability of meta-FVGs. Using these types complex field values can be generated without writing a single line of java code.

5.2.1 Post-Processing Meta Field Value Generators

FormattedNumberGenerator takes a FVG producing numbers and a format-element string. It formats the output of its FVG number according to the text in the format-element. Figure 4 shows an example, which is explained below.

PaddingGenerator left or right pads a string value, generated by a FVG, to a fixed field width using the character indicated in the character-element parameter, e.g. WHITESPACE. If this meta-FVG pads to left or right is determined by the padToLeft-element, which accepts a boolean value (true, false)

UpperLowerCaseGenerator modifies string values, generated by its FVG, either into uppercase characters or lowercase characters. Depending on the content in the mode-element parameter, it modifies the entire string into uppercase, lowercase or first character uppercase, remaining string lowercase

FormulaGenerator makes it possible to use the output of other FVGs in mathematical and logical formulas. This generator is, for example, used when prices before and after tax or discount have to be computed.

5.2.2 Flow Control Meta Field Value Generators

ProbabilityGenerator has a list of probability-elements, each containing a FVG. Each probability-element also has a value-attribute defining the likelihood of running the corresponding FVG. So in fact this meta-FVG chooses which of the containing FVGs to run with the given likelihood. The value- attribute is a double, and the value-attributes of all listed probability-elements must add to 1.0. Figure 5 shows an example usage of this meta-FVG.

SequentialGenerator contains FVGs or meta-FVGs. They are

```
<field name="phonenumber">
  <type>java.sql.types.VARCHAR</type>
  <size>30</size>
  <generator name="FormattedNumberGenerator">
    <generator name="LongGenerator">
      <min>10010001</min>
      <max>9999999999</max>
    </generator>
  <format>(%d%d%d) %d%d%d-%d%d%d%d</format>
  </generator>
</field>
```

Figure 4: Definition of a new field containing a phone number

simply run one after another. For this to work as desired, most meta-FVGs are defining their generator-element as optional. So if a meta-FVG is run without any FVG definition, it works not on a FVG output, but on the current field value. This enables us to use a FVG at the beginning of a sequence to generate the basic field value followed by any number of meta-FVGs to change the generated value. Figure 5 shows a usage example.

SwitchGenerator is the equivalent to the java switch statement. It has one generator-element which gets the field value to check, and multiple case-elements, each with a value-attribute and a FVG. The fetched field value from the first FVG is checked against the value-attributes of the case-elements, and if they match, the corresponding FVG is run. There is also a default-element containing a FVG, which is run when no value-attribute of a case-element matched the fetched field value.

ReferenceGenerator is the main tool to solve inter-table and inter-table dependencies. Although the generator differs in the configuration and the referenced generator is implicitly specified it is technically a metaFVG. With this generator fields in other tables or in the same table can be referenced and are recomputed. If the other field is in the same row the referenced value is cached instead to save the cost of re-computation. In order to make it possible to achieve consistency over multiple references to the same tuple (e.g. reference to first and last name), a "sameRowAs" flag can be specified.

5.2.3 Examples

Consider a new column with a phone number should be added to the example table in Figure 2. Figure 4 shows a field definition for this case. The LongGenerator generates a number between 10010001 and 9999999999, and the FormattedNumberGenerator maps the generated digits according to the format string, from right to left. If there is no digit left, it inserts a “0”. So the resulting values in this field are phone numbers between (001) 001-0001 and (999) 999-9999.

Now for a more complex example, assume that the schema needs about as many female names as male ones, the names should be uppercase, and the field has to padded to the maximum field size. In this case, the tree-like structure with generator elements shown earlier, or we could use the SequentialGenerator to get a more readable XML definition can be used. Figure 5 is showing the XML code for the modified name field. The SequentialGenerator just runs all containing (meta-) FVGs one after another. The first meta-FVG in the sequence is the ProbabilityGenerator, so this one must generate a value. It does this by defining two possible choices, each with a likelihood of 50% and each running DictList as FVG. They read names either from a dictionary file “female_names” or from “male_names”. After one of the two DictList FVGs created a value, UpperLowerCaseGenerator is next. It has a mode element, which configures it to set the string to uppercase. As this meta-FVG is

```
<field name="name">
  <type>java.sql.types.VARCHAR</type>
  <size>100</size>
  <generator name="SequentialGenerator">
    <generator name="ProbabilityGenerator">
      <probability value="0.5">
        <generator name="DictList">
          <file>dicts/female_names.dict</file>
        </generator>
      </probability>
      <probability value="0.5">
        <generator name="DictList">
          <file>dicts/male_names.dict</file>
        </generator>
      </probability>
    </generator>
    <generator name="UpperLowerCaseGenerator">
      <mode>uppercase</mode>
    </generator>
    <generator name="PaddingGenerator">
      <character> </character>
      <padToLeft>true</padToLeft>
    </generator>
  </generator>
</field>
```

Figure 5: New definition of the name field differencing between male and female names

the second in the sequence, it does not need to define a FVG to generate a value. It simply reads and modifies the value generated by the DictList FVG, so that the value is uppercase after the UpperLowerCaseGenerator run. The last meta-FVG in the sequence is the PaddingGenerator. It reads the uppercase string from the previously executed (meta-) FVGs and pads it using the character “ ” (whitespace) to the left up to the field size (100).

6. USE CASES

In this section we will discuss two actual use cases of PDGF. We will show how the introduction of meta-FVGs has reduced the number of different FVGs needed to generate a specific data set and explain the positive impact on the complexity of describing a data set.

6.1 TPC-H and SSB

One of our first non-trivial data sets was the TPC-H data set which we implemented to compare the performance of PDGF to DBGen [15]. TPC-H is a data warehousing benchmark that is widely used in industry and academia. The TPC-H data set consists of 8 tables with a total of 61 columns. Many of these columns are dense increasing numerical keys, dictionary based or random strings, or uniformly distributed numbers. However, some data has to be calculated based on other values or combines different types of data. The data generation is not trivial and therefore the TPC provides DBGen the data generation tool for TPC-H. For the first implementation of the PDGF version of the data generator for TPC-H meta-FVGs were not available yet. Therefore, separate FVGs had to be implemented for every field that did not fit the generic data generators. An example is *O_Clerk*, it is a string consisting of the string "Clerk#" and a 9 digit number with leading zeros, e.g. "Clerk#000095423". Although a FVG that could append a random number on a static string was already implemented another one was needed that would do the leading zeros as well. With meta-FVGs we were able to replace this special FVG with a generic construct of meta-FVGs. Overall, we could reduce the number of specialized FVGs within by only editing the config file within 2 hours from

26 to 10. The remaining FVGs were used for constructing complicated interrelationships between tables, which, however, can also be solved with meta-FVGs. Subsequently, we implemented a data generator for the Star Schema Benchmark [17], which uses a variation of the TPC-H data set. The basic configuration was done in one day and does not use any specialized FVGs, but resolves all dependencies and special generation requirements with meta-FVGs. The same is possible for the TPC-H data set.

6.2 ETL Benchmark

Another project that uses PDGF is the TPC's ETL benchmark initiative TPC-DI [21]. The project is still ongoing but it was the reason for the development of the concept of meta-FVGs in the first place. So in contrast to the TPC-H benchmark, we utilized meta-FVGs right from the start. The data set is much more complex than TPC-H and contains history keeping relations and change data captures. All of the tables contain references or are referenced by another table. Overall the data set consists of 20 tables and over 200 columns. Nevertheless, we have only 19 custom written FVGs for the implementation. Most of these again could be replaced but cover some corner cases that make them much faster than an meta-FVG based solution. The most used meta-FVG is the NullGenerator with 56 occurrences followed by the ProbabilityGenerator with 32 occurrences.

One downside of our meta-FVG approach that became apparent during the implementation of the ETL benchmark is the bigger and less readable schema configuration file. For the ETL benchmark it has over 1000 lines of XML, which was the reason for an improved, highly reduced specification language. Furthermore, an extensive error message system was implemented that points out errors in the configuration files with line numbers, context, suggestions for corrections as well as help. In future work, we will build graphical tools to automate the generation of the config files as well as imports and templates to further reduce the size and complexity of these documents.

7. RELATED WORK

There are several generic data generators, deployed both in scientific research and commercial products. Commercial generators include Red Gate SQL Data Generator [18], DTM Data Generator [5], and GS DataGenerator [8]. These are are to some extent configurable, but do not support the concept of meta-FVGs. Some research has gone into the direction of generating data from existing databases and thus scaling these up, e.g., the framework by Bruno and Chaudhuri [3]. Their approach resembles a framework for the specification and generation of databases that can model data distributions with rich intra- and inter-table correlations. Their concept is orthogonal to meta-FVGs and not yet supported in PDGF. However, sampling existing databases does not give the same flexibility and precision of data generation as meta-FVGs. Another example is MUDD, a multi dimensional data generator that was developed for the recently released TPC-DS benchmark. It is configurable since it operates on distribution files that can manually be edited [20]. A framework that uses a very similar data generation strategy to PDGF is Myriad [1]. It is, however, in an early stage and does not support the advanced features of PDGF. To the best of our knowledge, there is no other generic data generator that supports meta-FVGs and an equal set of features as PDGF.

8. CONCLUSION

In this paper, we presented the concept of meta field field value generators (meta-FVG), an extension to the concept of field value generators (FVG) of PDGF. The use of meta-FVGs eases the development of data generators, increases the flexibility in data generation schema design, and reduces implementation overhead. We formalized the relationship between FVG and meta-FVG and described their implementation in PDGF. To showcase the actual benefits of using meta-FVG, we present their use in three examples.

In future work, we will further reduce the complexity of specifying data generators using PDGF by implementing a graphical user interface for describing database schemas, their data content and data distribution. Furthermore, we will implement an automatic SQL generator that enables an easy database import of generated data. Finally, we will build a schema extractor that enables bootstrapping a generation schema from an existing commercial database with minimal interaction from the user.

A working version of PDGF with the SSB configuration is available at `http:\\www.paralleldatageneration.org`.

9. REFERENCES

[1] A. Alexandrov, K. Tzoumas, and V. Markl. Myriad: Scalable and Expressive Data Generation. In *VLDB'12*, 2012.

[2] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking Database Systems: A Systematic Approach. In *VLDB '83*, pages 8–19, 1983.

[3] N. Bruno and S. Chaudhuri. Flexible Database Generators. In *VLDB '05*, pages 1097–1107, 2005.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10*, pages 143–154, 2010.

[5] DTM Database Tools. DTM Data Generator. http://www.sqledit.com/dg/.

[6] M. Frank, M. Poess, and T. Rabl. Efficient Update Data Generation for DBMS Benchmark. In *ICPE '12*, 2012.

[7] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD '94*, pages 243–252, 1994.

[8] GSApps. GS Data Generator. http://www.gsapps.com/products/datagenerator/.

[9] J. E. Hoag and C. W. Thompson. A Parallel General-Purpose Synthetic Data Generator. *SIGMOD Record*, 36(1):19–24, 2007.

[10] K. Houkjær, K. Torp, and R. Wind. Simple and Realistic Data Generation. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246. VLDB Endowment, 2006.

[11] P. J. Lin, B. Samadi, A. Cipolone, D. R. Jeske, S. Cox, C. Rendón, D. Holt, and R. Xiao. Development of a Synthetic Data Set Generator for Building and Testing Information Discovery Systems. In *ITNG '06: Proceedings of the Third International Conference on Information Technology: New Generations*, pages 707–712, Washington, DC, USA, 2006. IEEE Computer Society.

[12] P. E. O'Neil. The Set Query Benchmark. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. 1993.

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

[14] M. Poess, T. Rabl, M. Frank, and M. Danisch. A PDGF Implementation for TPC-H. In *TPCTC '11*, 2011.

[15] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A Data Generator for Cloud-Scale Benchmarking. In *TPCTC '10*, pages 41–56, 2010.

[16] T. Rabl and M. Poess. Parallel data generation for performance analysis of large, complex RDBMS. In *DBTest '11*, page 5, 2011.

[17] T. Rabl, M. Poess, H.-A. Jacobsen, P. E. O'Neil, and E. O'Neil. Variations of the Star Schema Benchmark to Test Data Skew in Database Management Systems. In *ICPE '13*, 2013.

[18] Red Gate. SQL Data Generator 2.0. http://www.red-gate.com/products/sql-development/sql-data-generator/.

[19] SPC Benchmark 1TM Energy (SPC-1/E) Specification. http://www.storageperformance.org/specs/SPC-1_SPC-1E_v1.12.pdf.

[20] J. M. Stephens and M. Poess. MUDD: a multi-dimensional data generator. In *WOSP '04*, pages 104–109, 2004.

[21] L. Wyatt, B. Caufield, and D. Pol. Principles for an ETL Benchmark. In *TPC TC '09*, pages 183–198, 2009.