Hasso Plattner Institut
IT Systems Engineering
Universität Potsdam

Bachelor Thesis

Efficient Generation of Fully Configurable Structured Data

Andreas Henning

andreas.henning@student.hpi.uni-potsdam.de

Submitted on June 30th 2011

Supervised by: Prof. Dr. Felix Naumann

Abstract

For the development and evaluation of data analysis algorithms and approaches, it is crucial to apply suitable data sets for testing. However, most of the freely available data on the internet is not neccessarily suited for testing analysis algorithms. Additionally, adjusting existing open data to a certain analysis use case is very hard, because of the static nature of the data. In response to these problems, this work introduces a data generation tool that has been developed to allow the generation of freely configurable structured data, which can be used as a more suitable base for data analysis testcases. The functionality of this data generator includes the processing of user specified data structure files, which can contain multiple table definitions including basic schema information, keys, and functional dependency constraints. The generated data, based on these structures, can then be used to create better test cases for analysis algorithms, covering average test data as well as corner cases.

Zusammenfassung

Für die Entwicklung und Evaluierung von Datenanalysealgorithmen und -verfahren ist es wichtig, geeignete Datensätze zum Testen zu verwenden. Allerdings sind die frei im Internet verfügbaren Daten oftmals nicht für spezielle Anwendungen geeignet, und durch ihre statische Natur nur schwer auf zu lösende Probleme anzupassen. Im Zuge dieser Arbeit wurde daher ein Datengenerator entwickelt, der es ermöglichen soll, frei konfigurierbare Daten zu erzeugen, und damit den Analyseverfahren bessere Testdaten zugrunde legen zu können. Dabei werden vom Nutzer konfigurierte Schemadateien gelesen, die mehrere Tabellendefinitionen mit Schlüsselbeziehungen und funktionalen Abhängigkeiten enthalten können, und entsprechend strukturierte Daten zu erzeugen. Diese Daten können nun als Grundlage für die Evaluierung von Analyseverfahren verwendet werden, um gezieltere Testdaten, beispielsweise für Randfälle, zu erzeugen.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Potsdam, den 30.06.2011

Contents

1. Build Instructions

Building the *dbtesma* data generator is not difficult, but the required amount of preparation varies on different operating systems. The sections below provide installation instructions for the most common operating systems.

1.1. Build Instructions for Linux

To successfully build the *dbtesma* on Linux, the presence of the latest releases of `gcc`, `make` and `tar` is required. To compile the tool, issue the following command sequence:

```
1  mkdir dbtesma
2  cd dbtesma
3  wget http://misc.tonfeder.de/downloads/dbtesma/source_v1.0.4.tar
4  tar -xvf source_v1.0.4.tar
5  ./setup.sh
6  make
```

The binaries are now located in the `dbtesma` directory.

1.2. Build Instructions for Mac OS X

The steps required to successfully build the *dbtesma* on Mac OS X mirror those of the linux build, with the addition that the GNU versions of the mentioned tools are required. The proprietary Mac tools may have the same name and similar interfaces, but they might cause unintended behaviour during the compile process.

1.3. Build Instructions for Windows

There is a prebuilt Win32 NSIS installer available from
`http://misc.tonfeder.de/downloads/dbtesma/install_v1.0.4.exe`

To build the *dbtesma* from source on Windows machines anyway, the following steps have to be taken:

1. Download the source from
   `http://misc.tonfeder.de/downloads/dbtesma/source_v1.0.4.zip`
   and extract it.

2. Install the MinGW `gcc` port [2] and add the path to the `gcc` binary to the `PATH`.

3. Rename the `make` binary of the MinGW release to `make_old` (if it exists).

4. Install the GNU `make` untility [1] and add the path to the `make` binary to the `PATH`.

5. Issue the following command sequence within the `dbtesma` directory:

```
1  setup.bat
2  make
```

After these steps, the `dbtesma.exe` binary is located in the `dbtesma` directory of the source folder.

2. General Workflow of the Tool

The *dbtesma* data generator in its current state is designed as a command line based application. Therefore, all required parameters and configurations need to be passed as command line arguments or read from config files. We decided to follow this approach, because it allowed us to keep the user interaction model very simple, while recieving optimal configurability and reusability of data-structure configurations using plaintext files. Furthermore, all available GUI Frameworks are either platform dependent - forcing us to implement unnecessarily complex operation system dependent functionality - or require the installation of third party libraries, which we wanted to avoid.

For a full overview of all available command-line arguments, issue the following command:

```
1  ./dbtesma --help
```

Upon execution, the tool will parse the specified data configuration file, saving the structure configuration for the data-generation process. After the parsing is completed, this structure is being validated, returning errors to the user if necessary. During the validation process, the data-generation meta structure is refined, setting specified parameters and pointers to the required generation algorithms, whereupon the main data-generation process is initiated. During the data generation, each table is processed independently, generating values per column according to the stored generation methods, and writing data to the specified output file, one row at a time. The excessive preprocessing of the structure file allows us to design the time-consuming data generation very efficiently, reaching linear time complexity for the generation methods. Refer to section 5 for a detailed benchmark.

3. Tesmafile Specifications

The tesmafile is a human-readable plaintext file that containis the data structure utilized by the generator.

The following command yields an examplary tesmafile, that covers the basic functionality of the data generator:

```
./dbtesma --generate
cat tesmafile
```

Note that, similarily to Perl, the tesmafile parser uses the `#` character as a comment delimiter.

3.1. Basic Functionality

The *dbtesma* allows the creation of one schema per tesmafile, containing a specific set of tables and columns. This section demonstrates the configuration of a working schema with the most basic column and table attributes.

3.1.1. Defining a Table

An examplary table definition could look like this:

```
table={
  # required attributes
  name="tablex"
  rows="10000"

  [...]

}
```

Tables may only be defined at root level of the tesmafile, and not within other tables or columns. The required attribute `name` has to be unique in the table set, being used as filename for the resulting csv file.

Additionally, the `rows` attribute is required, representing the number of tuples in the table. Also, there must be at least one column defined within each table, indicated by `[...]` in the example above.

3.1.2. Defining a Column

An examplary column definition could look like this:

```
12  column={
13    # required attributes
14    name="columnx"
15    datatype="int"
16    length="11"
17
18    # optional attributes
19    unique="500"
20    basevalue="1"
21  }
```

The definition of a column must be located within a valid table. Similar to table names, the `name` attribute of a column has to be unique within each table. The `datatype` attribute must be one of $\{int|char|varchar\}$ and the `basevalue` attribute is required to be a valid element of the possible values for the column, specified by datatype and length. Currently, the *char* and *varchar* datatypes only support alphabets of $[a-z]$.

Additionally, the `length` and `basevalue` attributes for *int* columns are limited to the boundaries of the `unsigned long long` type of the compiler that has been used for the build. Even though integer overflows will not cause any runtime exceptions, they might lead to unintended structural behaviour.

3.2. Extended Functionality

In addition to the schema definition described above, it is possible to specify a range of different structures on the data within the tesmafile. This section shows how to apply these to the tesmafile.

3.2.1. Defining Primary Keys

Columns are declared as primary keys by adding `key="primary"` to their definition. A set of columns in the same table are declared a multicolumn primary key by additionally specifiying `key_group="x"` in every column, where x must be an integer. By using different values for x, any amount of independent multicolumn primary keys can be defined per table, where every column may only be part of one key group. Note that defining two key groups of size x and y in a single table, where $gcd(x, y) \neq 1$ will create interferences, resulting in more primary-key candidates than specified. This is a side effect of the used generation approach, and is described further in section 4.3.

3.2.2. Defining Foreign Keys

Columns are declared as foreign keys by adding `foreignkey="tablex:columnx"` to their definition, assuming `columnx` exists and is a primary key or primary key group column of `tablex`.

Multicolumn foreign keys are declared similarily, by specifying a set of columns as foreign keys to different columns of the primary key group. It is not necessary to cover all columns of the multicolumn primary key with foreign key columns, but recommended to have a reference to the leftmost column of the primary key group in the table definition. Note that only the foreign-key column referencing the leftmost primary key group column may be a calculated column; for example, by being located on the right-hand side of a functional dependency. Details of the limitations and implementation of the foreign key generation can be found in section 4.4.

3.2.3. Defining Functional Dependencies

Functional dependencies are declared by adding the following code to the table definition:

```
22  functional_dep={
23    lhs="columna,columnb"
24    rhs="columnx,columny"
25  }
```

Where `columna`, `columnb`, `columnx` and `columny` must be valid columns within the table definition the functional dependency is located in. The maximum amount of functional dependencies per table is limited to certain constraints. There may not be circular functional dependencies, and every column must only be part of the right-hand side columns of one functional dependecy. There is a sole exception from this rule: to allow simulation of multiple functional dependecies pointing to a single column. It is possible to define a set of functional dependencies, where the right-hand-side columns have exactly one column in common. But in this case, only one of the functional dependencies may contain left-hand-side columns which are referenced on the right-hand side of other functional dependencies. This problem is desribed further in section 4.5.5. More detailed information about the limitations and implementation of the functional dependency generation are located in section 4.5.

4. Algorithmic Approaches

To reach linear runtime complexity for the *dbtesma* data generator, we developed and implemented efficient algorithms for the different data structures and runtime procedures. Some of these approaches are described in this section.

4.1. Abstraction of Datatypes

All of the data-generation algorithms only operate on integers. To support more datatypes, we added a layer of abstraction. Every column owns an instance of the corresponding datatype-wrapper class, containing all the necessary datatype specific logic while implementing the same interface as any other datatype. Additionally, the value representation within the datatype wrappers is an integer, that indicates the numeric offset to the basevalue. This basevalue is specified in the tesmafile, or seeded with a random value. The offset is only converted to the actual value when the print method is called, allowing us to use the offsets as values for the calculation of other columns without much difficulty.

4.2. Abstraction of Generation Methods

Depending on the data structures a column belongs to, it may be necessary to calculate values from the previously written data of other columns. In that case, the references to the parent columns are stored within the child column during preprocessing of the structure file. During this preprocessing, the dbtesma decides which generation method to use for this column, storing a method pointer to the referenced generation method. This approach enables us to register the generation method once during preprocessing, in contrast to dealing with complicated if statements at each row of generated data, thus greatly improving generation speed.

4.3. Generation of Primary Keys

Single column primary keys are generated by incrementing the cell value at each row, generating a list of consecutive values ranged $[basevalue, basevalue + rows)$

To generate multicolumn primary keys, each value $p_{i,j}$ where i is the index of the column in the key group of size n and j is the number of the row to be written with i and j starting at 1, is calculated as: $p_{i,j} = \dfrac{basevalue_i + (j - 2 + i)}{n}$.

The following table is an example for $n = 4$ and $\forall i \in [1, n].basevalue_i = 1$

$j \backslash i$	1	2	3	4
1	1	1	1	1
2	1	1	1	2
3	1	1	2	2
4	1	2	2	2
5	2	2	2	2
6	2	2	2	3
7	2	2	3	3
8	2	3	3	3
9	3	3	3	3

However, there is one major issue with this approach, we have not yet been able to fix. When specifying two independent multicolumn primary keys of size n and m, where $gcd(n, m) > 1$, the generation method described above creates data that contains unintended false positives for primary-key candidates, due to interferences. This should be taken into account when specifying multiple multicolumn primary keys wihin a single table.

4.4. Generation of Foreign Keys

The foreign key values for a given column are random elements of a selected primary key column of any table. These are calculated by generating a random offset within the boundaries of the number of distinct values occurring in the primary key column. Since the basevalue of the primary key column is stored within the configuration structure, we extract this value, and use it as basevalue for our foreign key column. Adding the calculated offset per row to the seed will yield a random value, that is also contained in the primary key column, thus creating a single-column foreign key.

The Generation of multicolumn foreign keys is more difficult. Considering the structure of the generated multicolumn primary keys described in section 4.3, valid multicolumn-foreign-key tuples share a similar offset of ± 1 to the specified basevalues. Unfortunately, the data generation is handled in a cell-only scope, so the columns of the multicolumn foreign key have to reference each other. To accomplish this, we decided to declare one column of the multicolumn foreign key as a *head* column. Usually, the *head* column is the foreign key column referencing the leftmost column of the muticolumn primary key. This *head* column is taken as a parent for all of the following foreign key columns that reference columns of the same multicolumn primary key. In addition, each non-*head* foreign key column gets its left nighbour foreign key column as parent.

This results in the following generation rules:

1. **If the column is the** *head*, calculate a random value.

2. **If the column is not the** *head* **and has the** *head* **as sole parent**, assign the value of the *head* to this column and decide at random whether to increment the value by 1.

3. **If the column is not the** *head* **and has two columns as parents**, assign the value of the *head* to this column. If the second parents value is different from the *head* value, increment the value by 1; else, decide at random, whether to increment the value.

As a result of this approach, all columns of the multicolumn foreign key, except the *head*, must not be located on the right-hand side of functional dependencies. In addition to this, using the unique attribute on these columns is not allowed.

4.5. Generation of Functional Dependencies

Functional Dependencies are the most complex structure metrics currently included in the functionality of the dbtesma. This section is intended to clarify the idea inspiring the approach, as well as details of the implementation.

4.5.1. Conception

The naive approach to data generation including functional dependenceis, is to create lookup tables for every relation, storing written tuples and checking if they already occured, in which case a stored value needs to be written instead of the generated value. However, this approach needs a lot of main memory for the storage of the lookup tables, as well as computation time for the lookups and checks. In Addition, it is difficult to scale this approach on fully configurable systems of functional dependencies, because lots of references between the functional dependencies have to be resolved in order to ensure correct functional dependencies, which would require a very complex lookup table structure. In consideration of these facts, we decided to use a different approach.

Given a set of columns $\{A, B, C\}$ with the functional dependency $A, B \longmapsto C$ then this functional dependency is valid if and only if a function f exists, such that for every tuple $(a_i, b_i, c_i) \in ABC$ applies $f(a_i, b_i) = c_i$.
So, if we define a function f' and for every tuple $(a_i, b_i) \in AB$ caluculate $f'(a_i, b_i) = c'_i$, then $A, B \longmapsto C'$ is a valid functional dependency on ABC'.

In contrast to the naive approach, our implementation only generates a subset of all possible functional dependent data, because we use a fixed f, instead of allowing any function. However, this approach does not require us to store any lookup tables; therefore reducing the consumed amount of main memory and eliminating the need for expensive lookups and checks, while greatly improving scalabilty and ease of validation.

4.5.2. Limitation

When using the functional approach described in section 4.5.1, every column has to know the values of its parent columns in the same row. Thus, circular functional dependencies such as $A \longmapsto B$, $B \longmapsto C$, $C \longmapsto A$ can not be generated, because the parent requirement can not be met.

In addition, every column can only be registered as right-hand-side column once, because it is not possible to merge two separate generated values into one column - such that both individual functional dependencies are still valid - without storing context information or manipulating already generated data. This is illustrated in the example data below, where A, B, C, D are randomly filled columns, and the functional dependencies $A, B \longmapsto E, G$ and $C, D \longmapsto F, G$ are set:

A	B	C	D	E	F	G
2	3	5	7	1	4	x
11	17	19	23	9	14	y
2	3	9	10	1	6	x
11	17	9	10	9	6	x=y?

The first and second row of this sample are randomized data, mapping on different values of G. The third row introduces a new tuple for (C, D), but maps to the same value of G like the first row, because of the recurring tuple for (A, B). The fourth row, however, requires the value for G of row 2, following the recurring tuple for (A, B), but also requires the value for G of row 3, because of the recurring tuple for (C, D). Since the third row value of G is equal to the first row value of G, and the values of G in the first and second row are not equal, one of the given functional dependencies for G will break at this point.

Following these thoughts, we can conclude that there is no deterministic function f that will always generate valid values for G for given tuples (A, B, C, D), resulting in the limitation described. However, there is an exception from this rule, which will be explained in section 4.5.5

4.5.3. Validation

Since every right-hand-side column depends on the values of the corresponding left-hand-side columns, the order, in which the individual columns are generated, is important. To be able to generate the data in the right order, while not altering the order of the columns in the resulting csv files, the generation process and the output have been separated.

To determine the correct order of the columns, several steps are taken. First, every functional dependency is split into a number of independent functional dependencies containing only one right hand side column. For example, $A, B \longmapsto C, D$ would become $A, B \longmapsto C$ and $A, B \longmapsto D$. These are equivalent, but the second form is easier to handle in the following validation. Next, it is checked whether every column is only registered once as right hand side column.

The final step is a topological sort on the columns and their functional dependencies. A directed graph is created, that contains the columns of the current table as nodes and their functional dependencies as edges, where en edge $A \longmapsto B$ represents the statement "A is a parent column of B". Due to the nature of the topological sort, the resulting list of columns has the property that child columns are always generated after their parents, thus even including correct orders for transitive dependencies. If the topological sort fails, the graph contains a cycle, thus indicating an invalid configuration, which fits our requirements perfectly.

If the sort is successful, the left-hand-side columns of the individual functional dependencies are registered as parents to the affiliated right hand side columns, and the generation method of the right-hand-side columns is set accordingly.

4.5.4. Generation

Values for right-hand-side functional-dependency columns are calculated as $\frac{\sum_{i \in ParentValues}(i)}{ParentCount + 1}$

This method is a deterministic, non-injective function, thus creating a valid functional dependency. The non-injectivity is an important point, because an injective function would create a functional dependency, yet a reverse functional dependency simultaneously, which is not what we wanted.

4.5.5. Reverse Functional Dependencies

One of the limitations mentioned in 4.5.2, stated that it is impossible to generate data containing independent functional dependencies with the same right-hand-side columns, but this is not true in its entirety. We wanted to support at least a subset of these constellations, in case they might be needed for the evaluation of functional-dependency-analysis algorithms.

It is possible to declare multiple functional dependencies with common right-hand-side columns in a single table, if and only if the following requirements are met:

1. The functional dependencies have exactly one column in common on their right-hand sides.

2. Only one of the functional dependencies may have left hand side columns that are calculated by other means, e.g. being on the right-hand side of another functional dependency

If these points apply, then all but one of the functional dependencies are reversed, calculating the left-hand-side columns from the value of the right-hand-side column, thus simulating the correct functional dependency behaviour.

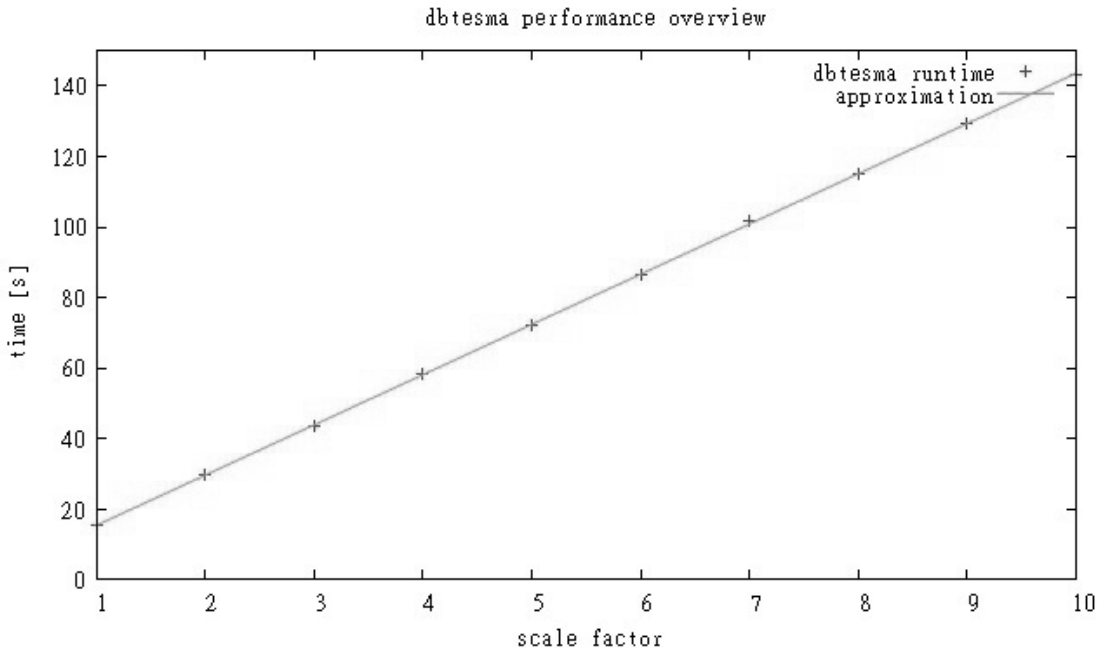
4.5.6. Hardened Functional Dependencies

Due to the nature of data generation, the values written into the tables are wide spread, thus likely to create unintended structure information. However, the functional dependency analysis performed during the project called for the generated data to be free of false positives for functional dependencies, to validate the efficiency of their algorithms. Following this requirement, we implemented a method to eliminate all false positives of functional dependencies from the generated data. To use this feature, the `--hardenFDs` switch has to be passed to the *dbtesma* executable on startup.

The actual elimintaion of the false positives is performed in a footer section of the generated data, where a number of tuples are generated that violate all non existant functional dependencies while leaving intended functional dependencies and other data structures intact.

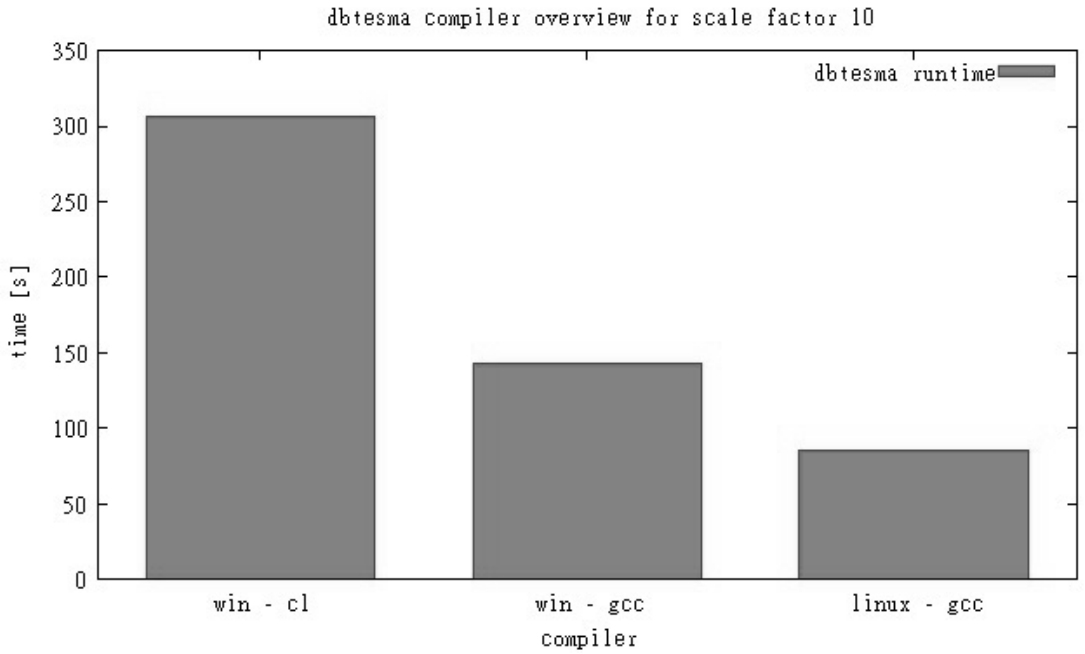
5. Benchmarking

The following illustration describes the performance of the *dbtesma* data generator on a Windows7 system running on an AMD Turion X2 2GHz with 2.5GB Ram, using the example schema with varying scale factors, where scale factor 1 represents a row count of 500 for table `extern_info` and a row count of 1000000 for table `main`. To get the best possible results, we executed the tool 10 times for each scale factor.



According to our benchmarking results, the graph of time to scale factor is approximately a linear function of the form $time = m * scalefactor + n$ where m is a constant factor representing the slope of the graph, and n is a number of seconds greater than zero, representing the duration of the table preprocessing, thus being a constant addition to all scale factors.

The resulting csv file for scale factor 10 has a size of approximately $828MB$, indicating an average duration of 176 seconds per Gigabyte of generated data. However, this time is significantly lower on recent systems, and about 40% reduced when the dbtesma is compiled and run on a linux machine of the same specifications. As a side note, the Microsoft Visual Studio compiler created binaries which took about 114% more time than the gcc compiled binaries on Windows, to process the same amount of data, which was one of the reasons why we decided to use a gcc port on Windows instead of the VS compiler.



The Figure above demonstrates the performance of the dbtesma after compilation with different compilers on the same computer, using the maxiumum speed optimization flags for each compiler. As before, every test has been run 10 times, to get optimal results.

6. Conclusion and Further Work

The *dbtesma* data generator in its current state is capable of creating data sets of considerable size in short time, while leaving the configuration of the structure of the data completely to the user. Many of the most important data structures, relevant for data analysis, are already directly supported, while the functionality that is still missing, for example distributions and inclusion dependencies, are covered by the sibling project [3], thus creating a complete data generation suite that efficiently creates fully configurable structured data.

However, we aspire to merge these generators, if possible, to a single tool, instead of using wrapper functionality. This will improve the ease of changing and adding functionality, thus increasing the maintainability and performance of the data generation, since time consuming communication and compatibility processes will be obsolete.

Furthermore, we want to continue the work on the data generator itself to refine the algorithms and the general implementation.

References

[1] *GNU Make for Windows.* `http://gnuwin32.sourceforge.net/packages/make.htm`,

[2] *MinGW | Minimalist GNU for Windows.* `http://www.mingw.org/`,

[3] Wolowyk, M.: Data Generation following specific Structures and Constraints. Inclusion Dependencies and statistic Distribution. (2011), June

A. Overview of Responsibilities by Classes

This list is intended to clarify the structure of the implementation of the dbtesma by providing a list of respoinsibilities for every class. For more detailed information about the structure of the classes, see the documentation inside the header files.

Naming convention: The namespaces of the classes represent the directory structure of the corresponding files. For example, Class `HELPER::Stringhelper` is declared in `dbtesma/HELPER/stringhelper.h` and defined in `dbtesma/HELPER/stringhelper.cpp`

Additionally to the classes, there are two files containing non-class code.

A.1. Non-Class Files

- `main.cpp`: contains the entry point of the application, as well as the top level processing logic, initiates command line parameter parsing, errors thrown during the preprocessig are returned to the user here
- `macros.h`: holds constant value definitions e.g. for the default tesmafile and the version string

A.2. Classes

- `HELPER::CliArgsHelper`: helper class responsible only for parsing the command line arguments, used by main.cpp
- `HELPER::FileHelper`: helper class responsible only for a small part of the file access handling, used by main.cpp to write the example tesmafile
- `HELPER::OsAbstractionHelper`: helper class responsible for anything that requires operating system dependent code, for example coloured output to stdout
- `HELPER::StringHelper`: helper class responsible for the string processing necessary for example during the tesmafile parsing, basically contains any functionality we needed on strings that was not available natively on std::string
- `HELPER::UiHelper`: helper class responsible for writing fancy responses back to stdout, including constant formatting

- `GEN::DataGenerator`: main data generation process class, used to contain a lot of the top level logic of the data generation process, but now only responsible for initiating the generation process on every table
- `CONF::Configparser`: customized context-based property file reader/parser, uses the functionality of `HELPER::StringHelper` excessively
- `CONF::Configvalidator`: validates attributes read by `CONF::Configparser`, doing lots of preprocessing on the tables and columns, including the registration of datatype wrappers, generation methods and the generation order of the columns. Additionally, error messages indicating invalid configurations are created here
- `DATA::Column`: data class representing a single column, contains generation logic per cell, actual output to csv is handled here, column scope information is also stored here
- `DATA::Config`: data class representing the whole config information, contains an array of tables, as well as pipelineing methods for the creation of new tables, columns and functional dependencies
- `DATA::Funcdep`: data class representing a single functional dependency, contains two arrays of columns to represent the right hand side and left hand side members, as well as pipelined accessors to these arrays
- `DATA::Funcdepgraph`: data class responsible for the topological sort on the columns, using the functional dependencies
- `DATA::Table`: data class representing a single table, contains table scope information, as well as medium level generation logic and the order information of the columns
- `DATA::WRAPPER::CharWrapper`: responsible for `char` related datatype processing
- `DATA::WRAPPER::DatatypeWrapper`: base class for datatype wrapper classes, provides interface
- `DATA::WRAPPER::IntWrapper`: responsible for `int` related datatype processing
- `DATA::WRAPPER::VarcharWrapper`: responsible for `varchar` related dataype processing