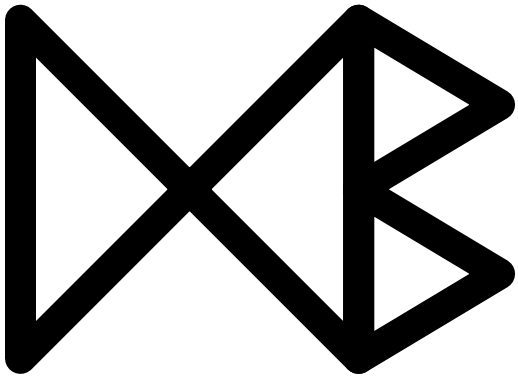
pygrametl: A Powerful Programming Framework for Extract–Transform–Load Programmers

Thomsen, Christian; Pedersen, Torben Bach



pygrametl: A Powerful Programming Framework for Extract–Transform–Load Programmers

Christian Thomsen and Torben Bach Pedersen

November, 2009

TR-25

A DB Technical Report

Title	pygrametl: A Powerful Programming Framework for Extract–Transform–Load Programmers
Author(s)	Christian Thomsen and Torben Bach Pedersen
Publication History	Extended version of: Christian Thomsen and Torben Bach Pedersen: "pygrametl: A Powerful Programming Framework for Extract–Transform–Load Programmers" in *Proceeding of the Twelfth International Workshop on Data Warehousing and OLAP*, Hong Kong, China, November 2009, pp. 49–56

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

Extract–Transform–Load (ETL) processes are used for extracting data, transforming it and loading it into data warehouses (DWs). Many tools for creating ETL processes exist. The dominating tools all use graphical user interfaces (GUIs) where the developer visually defines the data flow and operations. In this paper, we challenge this approach and propose to do ETL *programming* by writing code. To make the programming easy, we present the (Python-based) framework `pygrametl` which offers commonly used functionality for ETL development. By using the framework, the developer can efficiently create effective ETL solutions from which the full power of programming can be exploited. Our experiments show that when `pygrametl` is used, both the development time and running time are short when compared to an existing GUI-based tool.

1 Introduction

The Extract–Transform–Load (ETL) process is a crucial part for a data warehouse (DW) project. The task of the ETL process is to extract data from possibly heterogenous source systems, do transformations (e.g., conversions and cleansing of data) and finally load the transformed data into the target DW. It is well-known in the DW community that it is both time-consuming and difficult to get the ETL right due to its high complexity. It is often estimated that up to 80% of the time in a DW project is spent on the ETL.

Many commercial and open source tools supporting the ETL developers exist [2, 19]. The leading ETL tools provide graphical user interfaces (GUIs) in which the developers define the flow of data visually. While this is easy to use and easily gives an overview of the ETL process, there are also disadvantages connected with this sort of graphical programming of ETL programs. For some problems, it is difficult to express their solutions with the standard components available in the graphical editor. It is then time consuming to construct a solution that is based on (complicated) combinations of the provided components or integration of custom-coded components into the ETL program. For other problems, it can also be much faster to express the desired operations in some lines of code instead of drawing flows and setting properties in dialog boxes. In a recent article, Stodder names the impact of open source as a "BI megatrend". It is argued that "*[...] to customize data integration middleware through access to the source code is attractive because like it or not, many organizations need to tailor such routines to their requirements. Out-of-the-box routines only go so far.*" [18]. We agree with this but, in our opinion, this is also an argument that supports *programming* of ETL operations. It is also unattractive to integrate such a specialized routine into a GUI and provide the *eye-candy* like icons, configuration windows, etc. It is more productive only to program the core routine that does the data manipulation.

The productivity does not become high just by using a graphical tool. In fact, in personal communication with employees from a Danish company with a revenue larger than one billion US Dollars and hundreds of thousands of customers, we have learned that they gained no change in productivity after switching from hand-coding ETL programs in C to using one of the leading graphical ETL tools. Actually, the company experienced a decrease during the first project with the tool. In later projects, the company only gained the same productivity as when hand-coding the ETL programs. The main benefits were that the graphical ETL program provided standardization and self-documenting ETL specifications such that new team members easily could be integrated.

Trained specialists are often using textual interfaces efficiently while non-specialists use GUIs. In an ETL project, non-technical staff members often are involved as advisors, decision makers, etc. but the core development is (in our experience) done by dedicated and skilled ETL *developers* that are specialists. Therefore it is attractive to consider alternatives to GUI-based ETL programs. In relation to this, one can recall the high expectations to Computer Aided Software Engineering (CASE) systems in the eighties. It was expected that non-programmers could take part in software development by specifying (not programming) characteristics in a CASE system that should generate the code. Needless to say, the expectations were not fulfilled. It might be argued that forcing all ETL development into GUIs is a step back to the CASE idea.

We acknowledge that graphical ETL programs are useful in some circumstances but we also claim that for many ETL projects, a (completely or partly) code-based solution is the right choice. However, many parts of such code-based programs are redundant if each ETL program is coded from scratch. To remedy this, a framework with common functionality is needed.

In this paper, we present *`pygrametl`* (pronounced py-gram-e-t-l) which is a programming framework for ETL programmers. The framework offers functionality for ETL development and while it is easy to get an overview of and to start using, it is still very powerful. `pygrametl` offers a novel approach to ETL programming by providing a framework which abstracts the access to the underlying DW tables and allows the developer to

use the full power of the host programming language. In particular, the use of snowflaked dimensions is easy as the developer only operates on *one* "dimension object" for the entire snowflake while `pygrametl` handles the different DW tables in the snowflake. It is also very easy to insert data into dimension and fact tables while only iterating the source data once and to create new (relational or non-relational) data sources. Our experiments show that `pygrametl` indeed is effective in terms of development time and efficient in terms of performance when compared to a leading open-source GUI-based tool.

In previous work [20], we have been involved in building ETL tools for a DW that stores data about web resources and tests of these web resources. Experiences from this work have been used in building `pygrametl`. In particular, we have made it easy to insert data into dimensions and fact tables while only iterating through the source data once, to insert data into snowflaked dimensions that span several underlying tables, and to add new kinds of (relational or non-relational) data sources. Due to the ease of programming (we elaborate in Section 3) and the rich libraries, we implemented `pygrametl` as an application library in Python.

`pygrametl` is a framework where the developer makes the ETL program by coding it. `pygrametl` applies both functional and object-oriented programming to make the ETL development easy and provides often needed functionality. In this sense, `pygrametl` is related to other special-purpose frameworks where the user does coding but avoids repetitive and trivial parts by means of libraries that provide abstractions. This is, for example, the case for the web frameworks Django [1] and Ruby on Rails [14] where development is done in Python and Ruby code, respectively.

Many commercial ETL and data integration tools exist [2]. Among the vendors of the most popular products, we find big players like IBM, Informatica, Microsoft, Oracle, and SAP [4, 5, 7, 8, 15]. These are also the vendors named as the market leaders in Gartner's Magic Quadrant [2]. These vendors and many other provide powerful tools supporting integration between different kinds of sources and targets based on graphical design of the processes. Due to their wide field of functionality, the commercial tools often have steep learning curves and as mentioned above, the user's productivity does not necessarily get high(er) from using a graphical tool. Many of the commercial tools also have high licensing costs.

Open source ETL tools are also available [19]. In most of the open source ETL tools, the developer specifies the ETL process either by means of a GUI or by means of XML. Scriptella [16] is an example of a tool where the ETL process is specified in XML. This XML can, however, contain embedded code written in Java or a scripting language. `pygrametl` goes further than Scriptella and does not use XML around the code. Further, `pygrametl` offers DW-specialized functionality such as direct support for slowly changing dimensions and snowflake schemas, to name a few.

The academic community has also been attracted to ETL. A recent paper [22] presents a survey of the research. Most of the academic approaches, e.g., [17, 21], use UML or graphs to model an ETL workflow. In this paper, we challenge the idea that graphical programming of ETL is always easier than text-based programming. Grönniger et al. [3] have previously argued why text-based modeling is better than graphical modeling. Among other things, they point out that writing text is more efficient than drawing models, that it is easier to grasp details from text, and that the creative development can be hampered when definitions must be added to the graphical model. As graphical ETL tools often are model-driven such that the graphical model is turned into the executable code, these concerns are, in our opinion, also related to ETL development. Also, Petre [10] has previously argued against the widespread idea that graphical notation and programming always lead to more accessible and comprehensible results than what is achieved from text-based notation and programming. In her studies [10], she found that text overall was faster to use than graphics.

The rest of this paper is structured as follows: Section 2 presents an example of an ETL scenario which is used as a running example in the paper. Section 3 gives an overview of `pygrametl`. Sections 4–8 present the functionality and classes provided by `pygrametl`. There are classes that represent different *data sources* (described in Section 4), *dimensions* (Section 5), *fact tables* (Section 6), and *steps* in an ETL flow (Section 7). Further, there are a number of convenient helper functions (Section 8) that provide often needed functionality. Section 9 evaluates `pygrametl`. Finally, Section 10 concludes and points to future work.

2 Example Scenario

In this section, we describe an ETL scenario which we use as a running example throughout the paper. The example considers a DW where test results for tests of web pages are stored. This is inspired by the work we did

Table 1: The source data format for the running example

Field	Explanation
localfile	Name of local file where the page was stored
url	URL from which the page was downloaded
server	HTTP header's Server field
size	Byte size of the page
downloaddate	When the page was downloaded
lastmoddate	When the page was modified

(a) DownloadLog.csv

Field	Explanation
localfile	Name of local file where the page was stored
test	Name of the test that was applied to the page
errors	Number of errors found by the test on the page

(b) TestResults.csv

in the European Internet Accessibility Observatory (EIAO) project [20] but has been simplified here for the sake of brevity.

In the system, there is a web crawler that downloads web pages from different web sites. Each downloaded web page is stored in a local file. The crawler stores data about the downloaded files in a download log which is a tab-separated file. The fields of that file are shown in Table 1(a).

When the crawler has downloaded a set of pages, another program performs a number of different tests on the pages. These tests could, e.g., test if the pages are *accessible* (i.e., usable for disabled people) or conform to certain standards. Each test is applied to all pages and for each page, the test outputs the number of errors detected. The results of the tests are also written to a tab-separated file. The fields of this latter file are shown in Table 1(b).

After all tests are performed, the data from the two files is loaded into a DW by an ETL program. The schema of the DW is shown in Figure 1. The DW schema has three dimensions: The test dimension holds information about each of the tests that are applied. This dimension is static and prefilled (and not changed by the ETL). The date dimension holds information about dates and is filled by the ETL on-demand. The page dimension is *snowflaked* and spans several tables. It holds information about the individual downloaded web pages including both static aspects (the URL and domain) and dynamic aspects (size, server, etc.) that may change between two downloads. The page dimension is also filled on-demand by the ETL. The page dimension is a so-called *slowly changing dimension* where information about different *versions* of a given web page is stored.

Each dimension has a surrogate key (with a name ending in "id") and one or more attributes. The individual attributes have self-explanatory names and will not be described in further details here. There is one fact table which has a foreign key to each of the dimensions and a single measure holding the number of errors found for a certain test on a certain page on a certain date.

3 Overview of the Framework

The purpose of `pygrametl` is to make it easy to load data into DWs managed by relational database managements systems (RDBMSs). The trend on the commercial market for ETL is moving towards big suites of integration tools [2] supporting many kinds of targets. Focusing on RDBMSs as the targets for `pygrametl` keeps the design simple as it allows us to make assumptions and go for the good solutions specialized for this domain instead of thinking in very general "integrations terms". The data sources do *not* have to be relational.

When using `pygrametl`, the programmer makes code that controls the flow, the extraction (the E in ETL) from source systems, the transformations (the T in ETL) of the source data, and the load (the L in ETL) of the transformed data. For the flow control, extraction, and load, `pygrametl` offers components that support the developer and it is easy for the developer to create more of these components. For the transformations, the programmer benefits from having access to the full-fledged host programming language.

The loading of data into the target DW is particularly easy with `pygrametl`. The general idea is that the programmer creates *objects* for each *fact table* and *dimension* (different kinds are directly supported) in the DW. An object representing a dimension offers convenient methods like `insert`, `lookup`, etc. that hide all details of

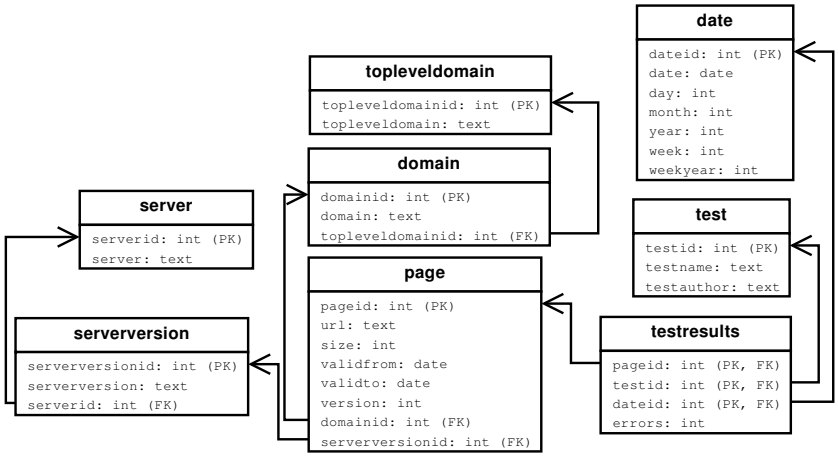


Figure 1: The schema for the running example.

caching, key assignment, SQL insertion, etc. In particular it should be noted that a snowflaked dimension also is treated in this way such that a single object can represent the entire dimension but the data is inserted into several tables in the underlying database.

The dimension object's methods take *rows* as arguments. A row in `pygrametl` is simply a mapping from names to values. Based on our personal experiences with other tools, we found it important that `pygrametl` does not try to validate that all data rows given to a dimension object have the same attributes or the same attribute types. If the programmer wants such checks, (s)he should make code for that. It is then, e.g., possible for the programmer to leave an attribute that was used as temporary value holder in a row or on purpose to leave out certain attributes. Only when attributes needed for `pygrametl`'s operations are missing, `pygrametl` complains. Attribute values that should be inserted into the target DW must exist when the insertion is done as `pygrametl` does not try to guess missing values. However, `pygrametl` has functionality for setting default values and/or on-demand call-back of user-defined functions that provide the missing values. Some other existing tools are strict about enforcing uniformity of rows. In `pygrametl`, it should be easy for the programmer to do what (s)he wants – not what the tool *thinks* (s)he wants.

`pygrametl` is implemented as a module in Python [13]. Many other programming languages could obviously have been used. We chose Python due to its design to support programmer productivity and its comprehensive standard libraries. Further, Python is both dynamically typed (the programmer does not have to declare the type a variable takes) and strongly typed (if a variable holds an integer, the programmer cannot treat it like a string). Consider, for example, this function:

```
def getfloat(value, default=None):
    try:
        return float(value)
    except Exception:
        return default
```

This function converts its input to a float or – if the conversion fails – to another value which defaults to None, Python's null value. Note that no types are specified for the input variables in the function declaration. It is possible to call the function with different types as in the following:

```
f1 = getfloat(10)
f2 = getfloat('1e1')
f3 = getfloat('A string', 10.0)
f4 = getfloat(['A', 'list'], 'Not a float!')
```

After this, `f1`, `f2`, and `f3` all equal 10.0 while `f4` holds the string 'Not a float!'. The expression `f1 + f2` will thus succeed, while `f3 + f4` will fail since a float and a string cannot be added.

Python is object-oriented but to some degree it also supports functional programming, e.g., such that functions or lambda expressions can be used as arguments. This makes it very easy to customize behavior. `pygrametl`,

for example, exploits this to support calculation of missing values on-demand (see Section 5). As Python also supports default arguments, pygrametl provides reasonable defaults for most arguments to spare the developer for unnecessary typing.

4 Data Source Support

In this and the following sections, we describe the functionality provided by pygrametl. As explained in Section 3, data is moved around in *rows* in pygrametl. Instead of implementing our own row class, we use Python's built-in dictionaries that provide efficient mappings between keys (i.e., attribute names) and values (i.e., attribute values). The data sources in pygrametl pass data on in such dictionaries. Apart from that, the only requirement to a data source is that it is iterable (i.e., its class must define the __iter__ method) such that code as the following is possible: **for** row **in** datasrc:.... Thus, it does not require a lot of programming to create new sources (apart from the code that does the real extraction which might be simple or not depending on the source format). For typical use, pygrametl provides a few, basic data sources described below.

SQLSource is a data source returning the rows of an SQL query. The query, the database connection to use and optionally new names for the result columns and "initializing" SQL are given when the data source is initialized.

CSVSource is a data source returning the lines of a delimiter separated file turned into dictionaries. This class is in fact just implemented in pygrametl as a reference to the class csv.DictReader in Python's standard library. Consider again the running example. There we have two tab-separated files and one instance of CSVSource should be created for each of them to load the data. For TestResults.csv, this is done as in

```
testresults = CSVSource(file('TestResults.csv', 'r'),
                        delimiter='\t')
```

Again, we emphasize the flexibility of using a language like Python for the pygrametl framework. Much more configuration can be done during the instantiation than what is shown but default values are used in this example. The input could also easily be changed to come from another source than a file, e.g., a web resource or a string in memory.

MergeJoiningSource is a data source that equijoins rows from two other data sources. It is given two data sources (which must deliver rows in sorted order) and information about which attributes to join on. It then merges the rows from the two sources and outputs the combination of the rows.

In the running example, we consider data originating from two data sources. Both the data sources have the field localfile and this is how we relate information from the two files:

```
inputdata = MergeJoiningSource(testresults, 'localfile',
                               downloadlog, 'localfile'))
```

where testresults and downloadlog are CSVSources.

HashJoiningSource is also a data source that equijoins rows from two other data sources. It does this by using a hash map. Thus, the input data sources do not have to be sorted.

5 Dimension Support

In this section, we describe the classes representing dimensions in the DW to load. This is the area where the flexibility and easy use of pygrametl are most apparent. Figure 2 shows the class hierarchy for the dimension supporting classes. Methods only used internally in the classes and attributes are not shown. Only required arguments are shown, not those that take default values when not given. Note that SnowflakedDimension actually does not inherit from Dimension but offers the same interface and can be used as if it were a Dimension due to Python's dynamic typing.

5.1 Basic Dimension Support

Dimension is the most basic class for representing a DW dimension in pygrametl. It is used for a dimension that has exactly one table in the DW. When an instance is created, the name of the represented dimension (i.e.,

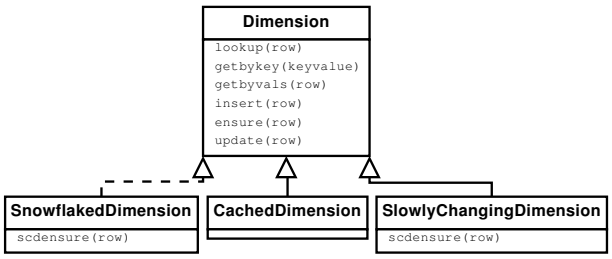


Figure 2: Class hierarchy for the dimension supporting classes.

the name of the table in DW), the name of the key column[1], and a list of attributes (the underlying table may have more attributes but `pygrametl` will not use them) must be given. Further, a number of optional settings can be given as described in the following. A list of *lookup attributes* can be given. These attributes are used when looking up the key value. Consider again the running example. The test dimension has the surrogate key testid but when data is inserted from the CSV files, the test in question is identified from its name (testname). The ETL application then needs to find the value of the surrogate key based on the test name. That means that the attribute testname is a lookup attribute. If no lookup attributes are given by the user, the full list of attributes (apart from the key) is used.

When the dimension object is given a row to insert into the underlying DW table (explained below), the row does not need to have a value for the dimension's key. If the key is not present in the row, a method (called `idfinder`) is called with the row as an argument. Thus, when creating a `Dimension` instance, the `idfinder` method can also be set. If not set explicitly, it defaults to a method that assumes that the key is numeric and returns the current maximum value for the key incremented by one.

A default key value for unfound dimension members can also be set. If a lookup does not succeed, this default key value is returned. This is used if new members should not be inserted into the dimension but facts still should be recorded. By using a default key value, the facts would then reference a prefilled member representing that information is missing. In the running example, test is a prefilled dimension that should not be changed by the ETL application. If data from the source file TestResults.csv refers to a test that is not represented in the test dimension, we do not want to disregard the data by not adding a fact. Instead, we set the default ID value for the test dimension to be -1 which is the key value for a preloaded dimension member with the value "Unknown test" for the testname attribute. This can be done as in the following code.

```
testdim = Dimension(name='test',
                    key='testid',
                    defaultidvalue=-1,
                    attributes=['testname', 'testauthor'],
                    lookupatts=['testname'])
```

Finally, it is possible for the developer to assign a function to the argument `rowexpander`. With such a function, it is in certain situations (explained below) possible to add required fields on-demand to a row before it is inserted into the dimension.

Many of the methods defined in the `Dimension` class accept an optional *name mapping* when called. This name mapping is used to map between attribute names in the rows (i.e., dictionaries) used in `pygrametl` and names in the tables in the DW. Consider again the running example where rows from the source file TestResults.csv have the attribute test but the corresponding attribute in the DW's dimension table is called testname. When the `Dimension` instance for test in `pygrametl` is given a row r to insert into the DW, it will look for the value of the testname attribute in r. However, this value does not exist since it is called test in r. A name mapping $n = \{$'testname' : 'test'$\}$ can then be set up such that when `pygrametl` code looks for the attribute testname in r, test is actually used instead. Examples showing the use of the methods are given in Section 9

`Dimension` offers the method `lookup` which based on the lookup attributes for the dimension returns the key for the dimension member. As arguments it takes a row (which at least must contain the lookup attributes) and optionally a name mapping. `Dimension` also offers the method `getbykey`. This method is the opposite

[1] We assume that a dimension has a non-composite key.

of `lookup`: As argument it takes a key value and it returns a row with all attributes for the dimension member with the given key value. Another method for looking up dimension members is offered by `Dimension`'s `getbyvals` method. This method takes a row holding a subset of the dimension's attributes and optionally a name mapping. Based on the subset of attributes, it finds the dimension members that have equal values for the subset of attributes and returns those (full) rows. For adding a new member to a dimension, `Dimension` offers the method `insert`. This method takes a row and optionally a name mapping as arguments. The row is added to the DW's dimension table. All attributes of the dimension must be present in the `pygrametl` row. The only exception to this is the key. If the key is missing, the `idfinder` method is applied to find the key value. The method `update` takes a row which must contain the key and one or more of the other attributes. The member with the given key value is updated to have the same values as the given attributes.

`Dimension` also offers a combination of lookup and insert: `ensure`. This method first tries to use `lookup` to find the key value for a member. If the member does not exist and no default key value has been set, `ensure` proceeds to use `insert` to create the member. In any case, `ensure` returns the key value of the member to the caller. If the `rowexpander` has been set (as described above), that function is called by `ensure` before `insert` is called. This makes it possible to add calculated fields before an insertion to the DW's dimension table is done. In the running example, the date dimension has several fields that can be calculated from the full date string (which is the only date information in the source data). However, it is expensive to do the calculations repeatedly for the same date. By setting `rowexpander` to a function that calculates them from the date string, the dependent fields are only calculated the first time `ensure` is invoked for certain date.

Compared to SQL Server Integration Services (SSIS) – a dominating ETL tool on the market – the functionality of `ensure` offers the programmer more flexibility. When using SSIS, the programmer should typically fill the dimension tables before fact tables are filled. The Lookup transformation cache in SSIS is filled before the data flow execution and it is hard to add new members on the fly such that they are available for later lookups. Workarounds are possible (e.g., use of stored procedures and hash tables for new members as in [23]) but they are complex and hard to maintain in comparison to the simple approach taken by `pygrametl` where the addition of new dimension members can be interleaved with the addition of facts. In our previous work in the EIAO project [20], it was only possible to run through the source data once due to time constraints and the used (RDF) data source and we had to add new dimension members when their first fact occurred. Such a strategy is very easy to implement with `pygrametl`.

`CachedDimension` has the same public interface as `Dimension` and the same semantics. However, it internally uses memory caching of dimension members to speed up lookup operations. The caching can be complete such that the entire dimension is held in memory or partial such that only the most recently used members are held in memory. A `CachedDimension` can also cache new members as they are being added. As noted above, addition of new dimension members to a cache is complex when using SSIS.

When an instance of `CachedDimension` is created, it is possible to set the same settings as for `Dimension`. Further, optional settings can decide the size of the cache, whether the cache should be prefilled with rows from the DW or be filled on-the-fly as rows are used, whether full rows should be cached or only keys and lookup attributes, and finally whether newly inserted rows should be put in the cache. In the running example, a `CachedDimension` for the test dimension can be made as in the following code.

```
testdim = CachedDimension(name='test',
                          key='testid',
                          defaultidvalue=-1,
                          attributes=['testname', 'testauthor'],
                          lookupatts=['testname'],
                          cachesize=500,
                          prefill=True,
                          cachefullrows=True)
```

5.2 Advanced Dimension Support

`SlowlyChangingDimension` provides support for type 1 and 2 changes in slowly changing dimensions [6]. When an instance of `SlowlyChangingDimension` is created, it can be configured in the same way as a `Dimension` instance. Further, the name of the attribute that holds versioning information for type 2 changes in the DW's dimension table should be set. A number of other things can optionally be configured. It is possible to set which attribute holds the "from date" telling from when the dimension member is valid. Likewise it is possible

to set which attribute holds the "to date" telling when a member becomes replaced. A default value for the "to date" for a new member can also be set. Further, functions that, based on the data in the rows, calculate these dates can be given but if they are not set, `pygrametl` defaults to use a function that returns the current date. `pygrametl` offers some convenient functions for this functionality. It is possible not to set any of these date related attributes such that no validity date information is stored for the different versions. It is also possible to list a number of attributes that should have type 1 changes (overwrites) applied. `SlowlyChangingDimension` has built-in cache support and its details can be configured.

`SlowlyChangingDimension` offers the same functions as `Dimension` (which it inherits) and the semantics of the functions are basically unchanged. `lookup` is, however, modified to return the key value for the *newest* version. To handle versioning, `SlowlyChangingDimension` offers the method `scdensure`. This method is given a row (and optionally a name mapping). It is similar to `ensure` in the sense that it first sees if the member is present in the dimension and, if not, inserts it. However, it does not only do a lookup. It also detects if any changes have occurred. If changes have occurred for attributes where type 1 changes should be used, it updates the existing versions of the member. If changes have also occurred for other attributes, it creates a new version of the member and adds the new version to the dimension. As opposed to the previously described methods, `scdensure` has side-effects on its given row: It sets the key and versioning values in its given row such that the programmer does not have to query for this information afterwards.

When a page is downloaded in the running example, it might have been updated compared to last time it was downloaded. To be able to record this history, we let the page dimension be a slowly changing dimension. We add a new version when the page has been changed and reuse the previous version when the page is unchanged. We lookup the page by means of the URL and detect changes by considering the other attributes. We create the `SlowlyChangingDimension` object as in the following.

```
pagedim = SlowlyChangingDimension(name='page',
                                  key='pageid',
                                  attributes=['url', 'size', 'validfrom', 'validto',
                                              'version', 'domainid' 'serverversionid'],
                                  lookupatts=['url'],
                                  fromatt='validfrom',
                                  fromfinder=pygrametl.datereader('lastmoddate'),
                                  toatt='validto',
                                  versionatt='version')
```

In the shown code, the `fromfinder` argument is a method that extracts a "from date" from the source data when creating a new member. It is also possible to give a `tofinder` argument to find the "to date" for a version to be replaced. If not given, this defaults to the `fromfinder`. If another approach is wished (e.g., such that the to date is set to the day before the new member's from date), `tofinder` can be set to a function which performs the necessary calculations.

`SnowflakedDimension` supports filling a dimension in a snowflake schema [6]. A snowflaked dimension is spread over more tables such that there is one table for each level in the dimension hierarchy. The fact table references one of these tables that itself references tables that may reference other tables etc. A dimension member is thus not only represented in a single table as each table in the snowflaked dimension represents a part of the member. The complete member is found by joining the tables in the snowflake.

Normally, it can be a tedious task to create ETL logic for filling a snowflaked dimension. First, a lookup can be made on the *root table* which is the table referenced by the fact table. If the member is represented there, it is also represented in the dimension tables further away from the fact table (otherwise the root table could not reference these and thus not represent the member at the lowest level). If the member is not represented in the root table, it must be inserted but it is then necessary to make sure that the member is represented in the next level of tables such that the key values can be used in references. This process continues for all the levels until the leaves[2]. While this is not difficult as such, it takes a lot of tedious coding and makes the risk of errors bigger. This is remedied with `pygrametl`'s `SnowflakedDimension` which takes care of the repeated `ensures` such that data is inserted where needed in the snowflaked dimension but such that the developer only has to make one method call to add/find the member.

An instance of `SnowflakedDimension` is constructed from other `Dimension` instances. The programmer creates a `Dimension` instance for each table participating in the snowflaked dimension and passes those

[2]It is also possible to do the lookups and insertions from the leaves towards the root but when going towards the leaves, it is possible to stop the search earlier if a part of the member is already present.

instances when creating the `SnowflakedDimension` instance. In the running example, the page dimension is snowflaked. We can create a `SnowflakedDimension` instance for the page dimension as shown in the following code (where different `Dimension` instances are created before).

```
pagesf = SnowflakedDimension([
                               (pagedim, [serverversiondim, domaindim]),
                               (serverversiondim, serverdim),
                               (domaindim, topleveldim)
                             ])
```

The argument is a list of pairs where a pair shows that its first element references each of the dimensions in the second element (the second element may be a list). For example, it can be seen that pagedim references serverversiondim and domaindim. We require that if t's key is named k, then an attribute referencing t from another table must also be named k. This requirement could be removed but it makes the specification of relationships between tables much easier. We also require that the tables in a snowflaked dimension form a tree (where the table closest to the fact table is the root) when we consider tables as nodes and foreign keys as edges. Again, we could avoid this requirement but this would complicate the ETL developer's specifications and the requirement does not limit the developer. If the snowflake does not form a tree, the developer can make `SnowflakedDimension` consider a subgraph that is a tree and use the individual `Dimension` instances to handle the parts not handled by the `SnowflakedDimension`. Consider, for example, a snowflaked date dimension with the levels day, week, month, and year. A day belongs both to a certain week and a certain month but the week and the month may belong to different years (a week has a week number between 1 and 53 which belongs to a year). In this case, the developer could ignore the edge between week and year when creating the `SnowflakedDimension` and instead use a single method call to ensure that the week's year is represented:

```
# Represent the week's year. Read the year from weekyear
row['weekyearid'] = yeardim.ensure(row,{'year':'weekyear'})
# Now let SnowflakedDimension take care of the rest
row['dateid'] = datesnowflake.ensure(row)
```

`SnowflakedDimension`'s `lookup` method calls the `lookup` method on the `Dimension` object for the root of the tree of tables. It is assumed that the lookup attributes belong to the table that is closest to the fact table. If this is not the case, the programmer can use `lookup` or `ensure` on a `Dimension` further away from the root and use the returned key value(s) as lookup attributes for the `SnowflakedDimension`. The method `getbykey` takes an optional argument that decides if the full dimension member should be returned (i.e., a join between the tables of the snowflaked dimension is done) or only the part from the root. This also holds for `getbyvals`. `ensure` and `insert` work on the entire snowflaked dimension starting from the root and moving outwards as much as needed. The two latter methods actually use the same code. The only difference is that `insert`, to be consistent with the other classes, raises an exception if nothing is inserted (i.e., if all parts were already there). Algorithm 1 shows how the code *conceptually* works but we do not show details like use of name mappings and how to keep track of if an insertion did happen. The algorithm is recursive and both `ensure` and `insert` first invoke it with the table $dimension$ set to the table closest to the fact table. On line 1, a normal

Algorithm 1 `ensure_helper`($dimension, row$)

```
1:  keyval ← dimension.lookup(row)
2:  if found then
3:      row[dimension.key] ← keyval
4:      return keyval
5:  for each table t that is referenced by dimension do
6:      keyval ← ensure_helper(t, row)
7:  if dimension uses the key of a referenced table as a lookup attribute then
8:      keyval ← dimension.lookup(row)
9:      if not found then
10:         keyval ← dimension.insert(row)
11: else
12:     keyval ← dimension.insert(row)
13: row[dimension.key] ← keyval
14: return keyval
```

`lookup` is performed on the table. If the key value is found, it is set in the row and returned (lines 2–4). If not, the algorithm is applied recursively on each of the tables that are referenced from the current table (lines 5–6). As side-effects of the recursive calls, key values are set for all referenced tables (line 3). If the key of one of the referenced tables is used as a lookup attribute for *dimension*, it might just have had its value changed in one of the recursive calls and a new attempt is made to look up the key in *dimension* (lines 7–8). If this attempt fails, we insert (part of) *row* into *dimension* (line 10). We can proceed directly to this insertion if no key of a referenced table is used as a lookup attribute in *dimension* (lines 11–12).

`SnowflakedDimension` also offers an `scdensure` method. This method can be used when the root is a `SlowlyChangingDimension`. In the running example, we previously created `pagedim` as an instance of `SlowlyChangingDimension`. When `pagedim` is used as the root as in the definition of `pagesf` above, we can use the slowly changing dimension support on a snowflake. With a single call of `scdensure`, a full dimension member can be added such that the relevant parts are added to the five different tables in the page dimension.

As previously mentioned, it is difficult to add dimension members and facts interleaved when using SSIS. The complexity of the workarounds to do this gets even higher when they must be applied to more tables as in snowflakes. In contrast it is very easy when using `pygrametl`. Also when using the open source graphical ETL tool Pentaho Data Integration (PDI), use of snowflakes requires the developer to use several lookup/update steps. It is then not possible to start looking up/inserting from the root as foreign key values might be missing. Instead, the developer has to start from the leaves and go towards the root. In `pygrametl`, the developer only has to use the `SnowflakedDimension` instance once. The `pygrametl` code considers the root first (and may save lookups) and only if needed moves on to the other levels.

6 Fact Table Support

`pygrametl` also offers three classes to represent fact tables. In this section, we describe these classes. It is assumed that a fact table has a number of key attributes and that each of these is referencing a dimension table. Further, the fact tables may have a number of measure attributes.

`FactTable` provides a basic representation of a fact table. When an instance is created, the programmer gives information about the name fact table, names of key attributes and optionally names of measure attributes. Note that the methods defined for `FactTable`s also support optional name mappings.

`FactTable` offers the method `insert` which takes a row and inserts a fact into the DW's table. This is obviously the most used functionality. It also offers a method `lookup` which takes a row that holds values for the key attributes and returns a row with values for both key and measure attributes. Finally, it offers a method `ensure` which first tries to use `lookup`. If a match is found on the key values, it compares the measure values between the fact in the DW and the given row. It raises an error if there are differences. If no match is found, it invokes `insert`. All the methods support name mappings.

`BatchFactTable` inherits `FactTable` and has the same methods. However, it does not insert rows immediately when `insert` is called but waits until a user-configurable number of rows are available. This can lead to a high performance improvement.

`BulkFactTable` provides a write-optimized representation of a fact table. It does offer the `insert` method but not `lookup` or `ensure`. When `insert` is called, the data is not inserted directly into the DW but instead written to a file. When a user-configurable number of rows have been added to the file (and at the end of the load), the content of the file is *bulkloaded* into the fact table.

The exact way to bulkload varies from DBMS to DBMS. Therefore, we again rely on Python's functional programming support and require the developer to pass a function when creating an instance of `BulkFactTable`. This function is invoked by `pygrametl` when the bulkload should take place. When using the database driver psycopg2 [12] and the DBMS PostgreSQL [11], the function can be defined as below.

```
def pgbulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):
    global connection # Opened outside this method
    cursor = connection.cursor()
    cursor.copy_from(file=filehandle, table=name,  sep=fieldsep, null=nullval,
                     columns=attributes)
```

Further, the developer can optionally define which separator and line-ending `BulkFactTable` uses and which file the data is written to before the bulkload. A string value used to represent NULLs can also be defined. For the running example, a `BulkFactTable` instance can be created for the fact table as shown below.

```
facttbl = BulkFactTable(name='testresults',
                        measures=['errors'],
                        keyrefs=['pageid', 'testid', 'dateid'],
                        bulkloader=pgbulkloader,
                        bulksize=5000000)
```

7 Flow Support

To make it possible to create components with encapsulated functionality and easily connect such components, `pygrametl` offers support for *steps* and flow of data between them. The developer can, for example, create a step for extracting data, a step for cleansing, a step for logging, and a step for insertion into the DW's tables. Each of the steps can be coded individually and finally the data flow between them can be defined. This borrows one of the good aspects from GUI-based ETL tools, namely that it easy to keep different aspects separated and thus to get an overview of what happens in a sub-task. In `pygrametl`, we combine the best of both worlds: The developer can benefit from the expressiveness and power of coding but also use encapsulation and the built-in flow support.

`Step` is the basic class for flow support. It can be used directly or as a base class for other step-supporting classes. The programmer can for a given `Step` set a *worker function* which is applied on each row passing through the `Step`. If not set by the programmer, the function `defaultworker` (which does not do anything) is used. Thus, `defaultworker` is the function inheriting classes override. The programmer can also determine to which `Step` rows by default should be sent after the current. That means that when the worker function finishes its work, the row is passed on to the next `Step` unless the programmer specifies otherwise. So if no default `Step` is set or if the programmer wants to send the row to a non-default `Step` (e.g., for error handling), there is the function `_redirect` which the programmer can use to explicitly direct the row to a specific `Step`.

There is also a method `_inject` for injecting a new row into the flow before the current row is passed on. The new row can be injected without an explicit target in which case the new row is passed on the `Step` that rows by default are sent to. The new row can also be injected and sent to a specified target. This gives the programmer a large degree of flexibility.

The worker function can have side-effects on the rows it is given. This is, for example, used in the class **`DimensionStep`** which calls `ensure` on a certain `Dimension` instance for each row it sees and adds the returned key to the row. Another example is **`MappingStep`** which applies functions to attributes in each row. A typical use is to set up a `MappingStep` applying `pygrametl`'s type conversion functions to each row. A similar class is **`ValueMappingStep`** which performs mappings from one value set to another. Thus, it is easy to perform a mapping from, e.g., country codes like 'DK' and 'DE' to country names like 'Denmark' and 'Germany'. To enable conditional flow control, the class **`ConditionalStep`** is provided. A `ConditionalStep` is given a condition (which is a function or a lambda expression). For each row, the condition is applied to the row and if the condition evaluates to a True value, the row is passed on to the next default `Step`. In addition, another `Step` can optionally be given and if the condition then evaluates to a False value for a given row, the row is passed on to that `Step`. Otherwise, the row is silently discarded. We emphasize how easy this is to use. The programmer only has to pass on a lambda expression or function. Also, to define new step functionality is very easy. The programmer just writes a single function that accepts a row as input and gives this function as an argument when creating a `Step`.

`Step`s can also be used for doing aggregations. The base class for aggregating steps, **`AggregatingStep`**, inherits `Step`. Like an ordinary `Step`, it has a `defaultworker`. This method is called for each row given to the `AggregatingStep` and must maintain the necessary data for the computation of the average. Further, there is a method `defaultfinalizer` that is given a row and writes the result of the aggregation to the row.

The functionality described above could also be implemented by the developer without `Steps`. However, if the developer prefers to think in terms of connected steps (as typically done in GUI-based ETL programs), (s)he can create specialized components with encapsulated functionality by using the `Step` classes. It is even possible to create a GUI from which the `pygrametl` `Steps` can be placed and connected visually while still allowing the programmer to take full advantage of Python programming.

8 Further Functionality

Apart from the classes described in the previous sections, `pygrametl` also offers some convenient methods often needed for ETL. These include functions that operate on rows (copy, rename, project, set default values) and functions that convert types, but return a user-configurable default value if the conversion cannot be done (like `getfloat` shown in Section 3).

In particular for use with `SlowlyChangingDimension` and its support for time stamps on versions, `pygrametl` provides a number of functions for parsing strings to create date and time objects. Some of these functions apply functional programming such that they dynamically create new functions based on their arguments. In this way specialized functions for extracting time information can be created. For an example, refer to `pagedim` we defined in Section 5. There we set `fromfinder` to a (dynamically generated) function that reads the attribute lastmoddate from each row and transforms the read text into a date object.

While this set of provided `pygrametl` functions is relatively small, it is important to remember that with a framework like `pygrametl`, the programmer also has access to the full standard library of the host language (in this case Python). Further, it is easy for the programmer to build up private libraries with the most used functionality.

9 Evaluation

To evaluate `pygrametl`, we implemented an ETL program for the running example. To get data, we made a data generator (details are given below). We also implemented an ETL solution in the graphical ETL tool Pentaho Data Integration (PDI) [9] to be able to compare the development efforts when using visual programming and code-based programming, respectively. PDI is a leading open-source ETL tool. It is Java-based and works with many different data sources and targets. Ideally, the comparison should have included commercial ETL tools but the license agreements of these tools (at least the ones we have read) explicitly forbid publishing of any evaluation/performance results without the consent of the provider. In this section, we present the findings of the evaluation. Note that `pygrametl`, the described ETL program, and the data generator are publicly available from `http://pygrametl.org`.

9.1 Development Time

We have previous experience with PDI but we obviously know the details of `pygrametl` very well. Therefore, it is hard to make a comprehensive comparison of the development times without having trained development teams at our disposal. We used each tool twice to create identical solutions. In the first use, we worked slower as we also had to find a strategy. In the latter use, we found the "interaction time" spent on typing and clicking.

The `pygrametl`-based ETL program was very easy to develop. It took a little less than one hour to code the complete ETL program in the first use. In the second use, it took 24 minutes. The program consists of 142 lines including comments and plenty of whitespace (for example, there is only one argument on each line when the `Dimension` objects are created). The program only has 56 Python statements. This strongly supports that it is easy to develop ETL programs using `pygrametl`. The main method of the developed ETL is shown below.

```
def main():
    for row in inputdata:
        extractdomaininfo(row)
        extractserverinfo(row)
        row['size'] = pygrametl.getint(row['size']) #Convert from string to int
        # Add the data to the dimension and fact tables
        row['pageid'] = pagesf.scdensure(row)
        row['dateid'] = datedim.ensure(row, {'date':'downloaddate'})
        row['testid'] = testdim.lookup(row, {'testname':'test'})
        facttbl.insert(row)
    connection.commit()
```

The methods `extractdomaininfo` and `extractserverinfo` have four lines of code that extract the domain, top-level domain, and server name from the URL and serverversion attributes. Note that the page dimension is a slowly changing dimension so we use `scdensure` to add new (versions of) members. This is a very easy way to fill a both snowflaked and slowly changing dimension. To fill the date dimension correctly, we have set a `rowexpander` for the `datedim` object to a function that (on demand) calculates the attribute values for a member to insert into the dimension. Thus, it is enough to use `ensure` to find or insert a member. The test dimension is preloaded and we only do lookups.

In comparison, the first PDI-based solution took us a little more than two hours to make work. In the second use of PDI, it took 28 minutes to create the solution. The solution has 19 boxes and 19 arrows between them. The flow is shown in Figure 3.

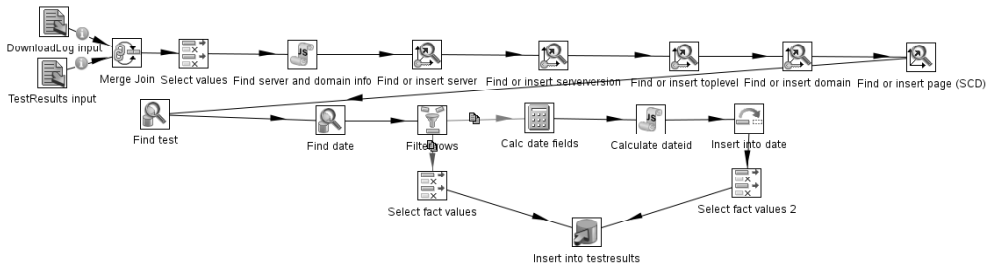


Figure 3: Data flow in PDI-based solution.

We emulate the `rowexpander` feature of `pygrametl` by first looking up a date and calculating the remaining date attributes in case there is no match. Note how we must fill the page snowflake from the leaves towards the root as discussed in Section 5.2.

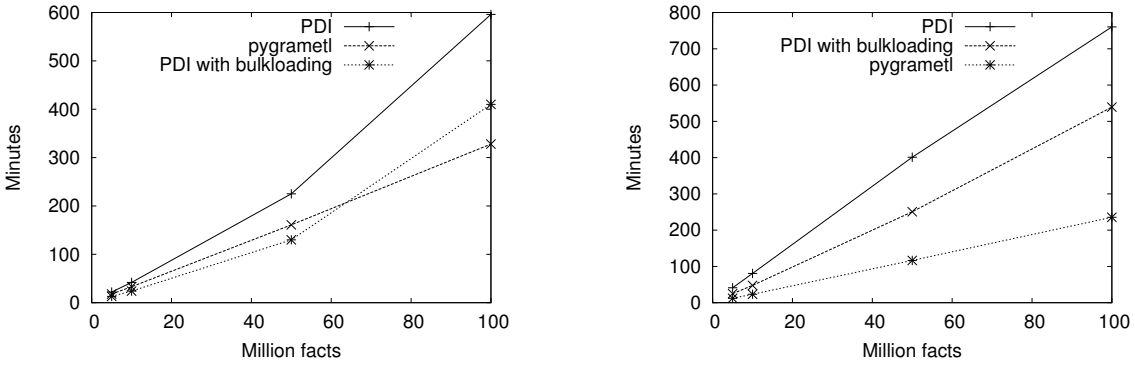
Comparing the number of statements (56) in the `pygrametl`-based solution to the number of boxes (19) in the PDI-based solution, it can be seen that one box in PDI corresponds to three Python statements when using `pygrametl`. This is a very promising result supporting that `pygrametl` is efficient to use (remember that a box is not just a box – rich dialogs must be used to configure its behaviour). From our experiments where we only measured the time spent on using keyboard and mouse, we found that the graphical tool was not faster to use than it was to program by code in a text editor. In fact, the typing of pure text was slightly faster.

In the experiment, `pygrametl` was faster to use than PDI in both uses. The first solution was much faster to create in `pygrametl` and we find that the strategy is far simpler to work out in `pygrametl` (compare the shown main method and Figure 3).

9.2 Performance

To test the performance of the solutions, we generated data. The generator was configured to create results for 2,000 different domains each having 100 pages. Five tests were applied to each page. Thus, data for one month gave 1 million facts. To test the SCD support, a page could remain unchanged between two months with probability 0.5. For the first month, there were thus 200,000 page versions and for each following month, there were ∼100,000 new page versions. We did the tests for 5, 10, 50, and 100 months, i.e., on data sets of realistic sizes. The solutions were tested on a single[3], powerful server with two quad-core 1.86GHz Xeon CPUs, 16GB of RAM,

[3] We did not test PDI's support for distributed execution.



(a) Elapsed time

(b) CPU time

Figure 4: Performance results.

and 10,000 rpm harddisks. The server ran SUSE Linux Enterprise Edition 10, Python 2.6, Sun Java 6SE, PDI 3.2-RC1, and PostgreSQL 8.3.5. We tested the tools on a DW where the primary key constraints were declared but the foreign key constraints were not. The DW had an index on `page(url, version)`. We loaded the data into DWs that already held data for 100 months.

PDI was tested in two modes. One with a single connection to the DW such that the ETL is transactionally *safe* and one which uses a special component for bulkloading the facts into PostgreSQL. This special component makes its own connection to the DW. This makes the load faster but transactionally *unsafe* as a crash can leave the DW loaded with inconsistent data. The `pygrametl`-based solution uses bulkloading (`BulkFactTable`) but is always running in *safe* transactional mode with a single connection to the DW. The solutions were configured to use caches without size limits. When PDI was tested, the max. Java heap size was set to 8GB (12GB for the largest data set as 8GB was too little for this).

Figure 4(a) shows the elapsed wall-clock time for the loads and Figure 4(b) shows the spent CPU time. It can be seen that the elapsed time grows super-linearly for PDI while it grows linearly for `pygrametl`. PDI does not scale linearly since swapping occurs for the big data set due to the high memory consumption (note that the spent CPU time does scale linearly for PDI).

PDI is the fastest when it is allowed to use two connections but this advantage fades for large data sets. In *safe* mode with a single transaction, the `pygrametl`-based solution is the fastest for all the data sets. When loading 100 million facts, the `pygrametl`-based solution handles 5081 facts/sec. PDI with a single connection handles 2796 and PDI with two connections handles 4065 facts/sec.

Servers may have many CPUs/cores but it is still desirable if the ETL uses little CPU time. More CPU time is then available for other purposes like processing queries. This is in particular relevant if the ETL is running on a virtualized server with contention for the CPUs. From Figure 4(b), it can be seen that `pygrametl` uses much less CPU time than PDI. When loading the data set with 50 million facts, `pygrametl`'s CPU utilization is 72%. PDI's CPU utilization is 177% with one connection and 191% with two. It is clearly seen that it in terms of resource consumption is beneficial to code a specialized light-weight program instead of using a general feature-rich but heavy-weight ETL application.

With a real-life, confidential data set (with around 8 millions facts in three fact tables and 8 dimensions of which one is a snowflake with two hierarchies and six participating tables), we have experienced that PDI 3.0 on the same server as before, loads the data set in around 240 minutes while a `pygrametl`-based solution uses around 100 minutes. The resource consumption of the `pygrametl`-based solution is also much smaller than PDI's. `pygrametl` used 70 minutes of CPU-time while PDI used 730 minutes of CPU time. This means that the average CPU usage for `pygrametl` was 70% while it was 300% for PDI (meaning that, on average, three cores were used concurrently by PDI). Further, `pygrametl` had 50MB resident in RAM while PDI had 3.3GB resident in RAM for that data set.

10 Conclusion and Future Work

In this paper, we presented `pygrametl` which is a programming framework for ETL programming. We challenged the conviction that ETL development is always best done in a graphical tool. We proposed to also let the ETL *developers* (that typically are dedicated experts) do ETL *programming* by writing code. Instead of "drawing" the entire program, the developers can concisely and precisely express the processing in code. To make this easy, `pygrametl` provides commonly used functionality such as data source access and filling of dimensions and fact tables. In particular, we emphasize how easy it is to fill snowflaked and slowly changing dimensions. A single method call will do and `pygrametl` takes care of all needed lookups and insertions.

`pygrametl` is implemented in Python. Python was chosen because it offers convenient functionality to boost the programming efficiency (including object-oriented and functional programming) and a rich standard library.

Our experiments have shown that ETL development with `pygrametl` is indeed efficient and effective. `pygrametl`'s flexible support of fact and dimension tables makes it easy to fill the DW and the programmer can concentrate on the needed operations on the data where (s)he benefits from the the power and expressiveness of a real programming language to achieve high productivity.

We do, however, acknowledge that some persons prefer a graphical overview of the ETL process. Indeed, an optimal solution could include both a GUI and code. In future work, we plan to make a GUI for creating and visually connecting steps. *Roundtrip engineering* such that updates in the code are visible in the GUI and vice versa should be possible. We also plan to investigate how to provide an efficient and yet simple way to create and run ETL programs in parallel or distributed DW environments. It should be possible to plug in new DW and ETL servers and have the coordination done automatically.

Acknowledgments

This work was supported by the Agile & Open Business Intelligence project co-funded by the Regional ICT Initiative under the Danish Council for Technology and Innovation under grant no. 07-024511.

References

[1] Django. `djangoproject.com/` as of 2009-06-18.

[2] T. Friedman, M.A. Beyer, and A. Bitterer. *Gartner Magic Quadrant for Data Integration Tools*, 2008.

[3] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Text-based Modeling. In *Proc. of ATEM*, 2007

[4] IBM InfoSphere DataStage. `ibm.com/software/data/ integration/datastage/` as of 2009-06-18.

[5] Informatica. `informatica.com` as of 2009-06-18.

[6] R. Kimball and M. Ross. *The Data Warehouse Toolkit*, 2nd Edition. Wiley, 2002.

[7] Microsoft SQL Server Integration Services. `microsoft.com/sqlserver/2005/en/us/integration-services.aspx` as of 2009-06-18.

[8] Oracle Warehouse Builder. `oracle.com/technology/products/warehouse/index.html` as of 2009-06-18.

[9] Pentaho Data Integration. `kettle.pentaho.org` as of 2009-06-18.

[10] M. Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Comm. ACM* 38(6):33-44, 1995.

[11] PostgreSQL. `postgresql.org` as of 2009-06-18.

[12] Psycopg2. initd.org/pub/software/psycopg/ as of 2009-06-18.

[13] Python. python.org as of 2009-06-18.

[14] Ruby on Rails. rubyonrails.org/ as of 2009-06-18.

[15] SAP BusinessObjects. sap.com/solutions/sapbusinessobjects/index.epx as of 2009-06-18.

[16] Scriptella. scriptella.javaforge.com as of 2009-06-18.

[17] A. Simitsis, P. Vassiliadis, M. Terrovitis, and S. Skiadopoulos. Graph-Based Modeling of ETL Activities with Multi-Level Transformations and Updates. In *Proc. of DaWaK* , pp. 43-52, 2005.

[18] D. Stodder. Nine BI Megatrends. *Intelligent Enterprise*, January 2009. Available from intelligententerprise.com/channels/business_intelligence/showArticle.jhtml?articleID=212700482 as of 2009-06-18.

[19] C. Thomsen and T.B. Pedersen. A Survey of Open Source Tools for Business Intelligence. *IJDWM* 5(3):56-75, 2009.

[20] C. Thomsen and T.B. Pedersen. Building a Web Warehouse for Accessibility Data. In *Proc. of DOLAP*, 2006.

[21] J. Trujillo and S. Luján-Mora. A UML Based Approach for Modeling ETL Processes in Data Warehouses. In *Proc. of ER2003*, pp. 307–320, 2003.

[22] P. Vassiliadis. A Survey of Extract–Transform–Load Technology. *IJDWM* 5(3):1-27, 2009.

[23] E. Veerman. Microsoft Server 2005. Project REAL: Business Intelligence ETL Design Practices, 2005. Available from http://technet.microsoft.com/en-us/library/cc966422.aspx as of 2009-06-18.