Programmatic ETL

Thomsen, Christian; Andersen, Ove; Jensen, Søren Kejser; Pedersen, Torben Bach

Published in:
Business Intelligence and Big Data - 7th European Summer School, eBISS 2017, Tutorial Lectures

DOI (link to publication from Publisher):
10.1007/978-3-319-96655-7_2

Publication date:
2018

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

Citation for published version (APA):
Thomsen, C., Andersen, O., Jensen, S. K., & Pedersen, T. B. (2018). Programmatic ETL. In E. Zimányi (Ed.), *Business Intelligence and Big Data - 7th European Summer School, eBISS 2017, Tutorial Lectures* (pp. 21-50). Springer Publishing Company. https://doi.org/10.1007/978-3-319-96655-7_2

Programmatic ETL

Christian Thomsen, Ove Andersen, Søren Kejser Jensen, Torben Bach Pedersen

Department of Computer Science, Aalborg University
{chr, xcalibur, skj, tbp}@cs.aau.dk

Abstract. Extract-Transform-Load (ETL) processes are used for extracting data, transforming it and loading it into data warehouses (DWs). The dominating ETL tools use graphical user interfaces (GUIs) such that the developer "draws" the ETL flow by connecting steps/transformations with lines. This gives an easy overview, but can also be rather tedious and require much trivial work for simple things. We therefore 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. In this chapter, we present our work on `pygrametl` and related activities. Further, we consider some of the lessons learned during the development of `pygrametl` as an open source framework.

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 [1,22]. 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.

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 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 chapter, we present *`pygrametl`* 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. For example, 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.

We started the work on `pygrametl` back in 2009 [23]. Back then, we had in a project with industrial partners been building a DW where a real-life dataset was loaded into a snowflake schema by means of a GUI-based ETL tool. It was apparent to us that the used tool required a lot of clicking and tedious work to be able to load the dataset. In an earlier project [21], we had not been able to find an ETL tool that fitted the requirements and source data. Instead we had created our ETL flow in Python code, but not in reusable, general way. Based on these experiences, we were convinced that the programmatic approach clearly was advantageous in many cases. On the other hand, it was also clear that the functionality for programmatic ETL should be generalized and isolated in a library to allow for easy reuse. Due to the ease of programming (we elaborate in Section 3) and the rich libraries, we chose to make a library in Python. The result was `pygrametl`. Since 2009 `pygrametl` has been developed further and made available as open source such that it now is used in proof-of-concepts and production systems from a variety of domains. In this chapter we describe the at the time of writing current version of `pygrametl` (version 2.5). The chapter is an updated and extended version of [23].

`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 [3] and Ruby on Rails [17] where development is done in Python and Ruby code, respectively.

Many commercial ETL and data integration tools exist [1]. Among the vendors of the most popular products, we find big players like IBM, Informatica, Microsoft, Oracle, and SAP [5,6,10,11,18]. 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 [22]. 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 [19] 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. Vassiliadis presents a survey of the research [28]. Most of the academic approaches, e.g., [20,26], use UML or graphs to model an ETL workflow. We challenge the idea that graphical programming of ETL is always easier than text-based programming. Grönniger et al. [4] 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 [13] 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 [13], she found that text overall was faster to use than graphics.

The rest of this chapter is structured as follows: Section 2 presents an example of an ETL scenario which is used as a running example. Section 3 gives an overview of `pygrametl`. Sections 4–7 present the functionality and classes provided by `pygrametl` to support *data sources*, *dimensions*, *fact tables*, and *flows*, respectively. Section 8 describes some other useful functions provided by `pygrametl`. Section 9 evaluates `pygrametl` on the running example. Section 10 presents support for parallelism in pygrametl and another pygrametl-based framework for MapReduce. Section 11 presents a case-study of a company using pygrametl. Section 12 contains a description of our experiences with making pygrametl available as open source. Section 13 concludes and points to future work. Appendix A offers readers who are not familiar with the subject a short introduction to data warehouse concepts.

2 Example Scenario

In this section, we describe an ETL scenario which we use as a running example. The example considers a DW where test results for tests of web pages are stored. This is inspired by work we did in the European Internet Accessibility Observatory (EIAO) project [21] 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).

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

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 *type 2 slowly changing dimension* [8] 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.

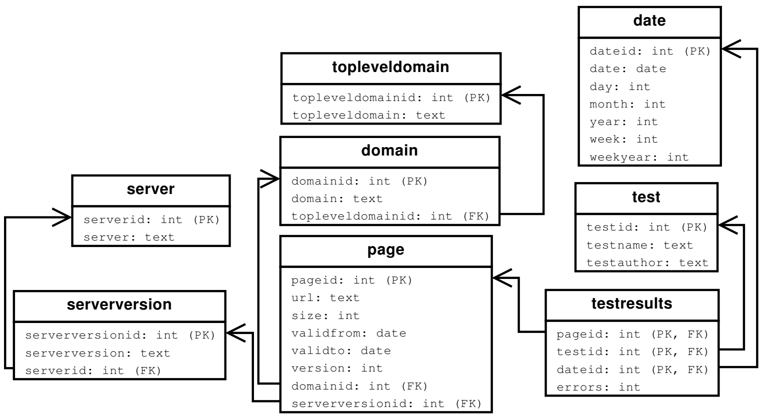


Fig. 1. The schema for the running example.

3 Overview of the Framework

Unlike many commercial ETL tools which can move data from sources to a variety of targets, the purpose of `pygrametl` is only to make it easy to load data into dimensional DWs [8] managed by relational database managements systems (RDBMSs). 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 "integration 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 Python 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 details of 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 although 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 [16]. 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 `pygrametl` 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 an instance of `CSVSource` should be created for each of them to load the data. For TestResults.csv, this is done as in

```
testresults = CSVSource(open('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.

The data sources described above were all available in the first public release of `pygrametl` in 2009. Since then, more sources have been added including the following.

`TypedCSVSource` is like a CSV source, but will perform type casts on (textual) values comming from the input file:

```
testresults = TypedCSVSource(open('TestResults.csv', 'r'),
                             casts={'size':int},
                             delimiter='\t')
```

While `TypedCSVSource` alweays overwrites an attribute value with the result of a function (a cast to an int in the example above), **`TransformingSource`** allows any transformation to be applied to a row, also transformations that add new attributes. For these two sources, Python's support for functional programming is used and functions are passed as arguments.

`CrossTabbingSource` can pivot data from another source. Data from other sources can also be filtered or unioned by **`FilteringSource`** and **`UnionSource`**, respectively. A **`DynamicForEachSource`** will for each given argument create a new source which is iterated by the `DynamicForEachSource` instance. This is, for example, useful for a directory with many CSV files to iterate. The user must provide a function that when called with a single argument returns a new source to iterate as exemplified below:

```
srcs = DynamicForEachSource([... sequence of the names of the files ...],
                            lambda f: CSVReader(open(f, 'r')))
for row in srcs: # will iterate over all rows from all the files;
   ...           # do something with the row data here
```

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 (classes for parallel load of dimensions are not shown for brevity). 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., the name of the table in DW), the

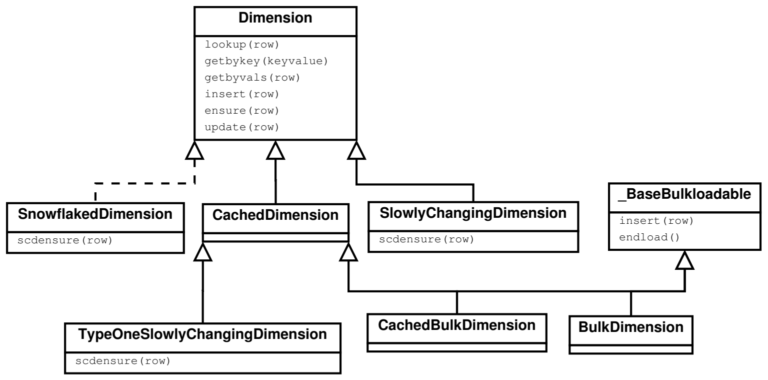


Fig. 2. Class hierarchy for the dimension supporting classes.

name of the key column[1], and a list of attributes (the underlying table may have more attributes but `pygrametl` will then 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'],
```

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

```
                    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.

`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 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.

`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.

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 2 changes in slowly changing dimensions [8] and in addition to type 2 changes, type 1 changes can also be supported for a subset of the dimension's attributes. 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 can be set. If it is set, the row version with the greatest value for this attribute is considered the newest row and `pygrametl` will automatically maintain the version number for newly added versions. If there is no such attribute with version information, the user can specify another attribute to be used for the ordering of row versions. 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 as well as a default value for the "from date" for the first version of a new member. Further, functions that, based on data in new rows to add, calculate the "to date" and "from date" 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. Finally, it is possible to configure if versions should be sorted by the DBMS such that `pygrametl` uses SQL's ORDER BY or if `pygrametl` instead should fix all versions of a given member and do a sort in Python. It depends on the used DBMS what performs the best, but for at least one popular commercial DBMS, it is significantly faster to let `pygrametl` perform the sorting.

`SlowlyChangingDimension` offers the same functions as `Dimension` (which it inherits from) 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 [8]. 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 `ensure`s 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 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 having 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

[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.

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 recur-

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

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

sive 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 `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.

When using the graphical ETL tools such as SQL Server Integration Services (SSIS) or the open source Pentaho Data Integration (PDI), use of snowflakes requires the developer to use several lookup/update steps. It is then not easily 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.

The previously described Dimension classes were all present in the first public release of `pygrametl` [23]. Since then more classes have been added. **`TypeOneSlowlyChangingDimension`** is similar to `SlowlyChaningDimension` apart from that it *only* suppors type 1 changes where dimension members are updated (not versioned) to reflect changes [8]. **`BulkDimension`** is used in scenarios where much data must be inserted into a dimension table and it becomes too time-consuming to use traditional SQL INSERTs (as the previously described `Dimension` classes do). `BulkDimension` instead writes new dimension values to a temporary file which can be bulk loaded.

The exact way to bulkload varies from DBMS to DBMS. Therefore, we rely on Python's functional programming support and require the developer to pass a function when creating an instance of `BulkDimension`. This function is invoked by `pygrametl` when the bulkload should take place. When using the database driver psycopg2 [15] and the DBMS PostgreSQL [14], 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)
```

The developer can optionally also define which separator and line-ending to use and which file the data is written to before the bulkload. A string value used to represent nulls can also be defined.

To enable efficient lookups, `BulkDimension` caches all data of the dimension in memory. This is viable for most dimensions as modern computers have big amounts of memory. In some scenarios it can, however, be infeasible to cache all data of a dimension in memory. If that is the case and efficient bulk loading still is desired, the **`CachedBulkDimension`** can be used. Like the `CachedDimension`, the size of its cache can be configured, but in addition it supports bulk loading. To avoid code duplication, code supporting bulk loading has been placed in the class **`_BaseBulkloadable`** which `BulkDimension` and `CachedBulkDimension` then inherit from (as does `BulkFactTable` described below).

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.

FactTable offers the method insert which takes a row (and optionally a name mapping) 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 instead keeps them in memory and 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.

BulkFactTable inherits from _BaseBulkloadable which provides support for bulk loading. As for BulkDimension and CachedBulkDimension, the user has to provide a function that invokes the bulk-loading method of her particular DB driver. 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

A good aspect from GUI-based ETL tools, is that it is easy to keep different aspects separated and thus to get an overview of what happens in a sub-task. To make it possible to create small 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.

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. This is very easy 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.

Steps 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 can of course also be implemented by the developer without Steps. However, the Step functionality was included in the first public release of pygrametl to support a developers who prefer to think in terms of connected steps (as typically done in GUI-based ETL programs). In hindsight, this seems to be a non-wanted functionality. While we have received comments, questions, and bug reports about most other areas of pygrametl, we have virtually never received anything about Steps. It seems that developers who choose to define their ETL flows with code, in fact prefer not to think in the terms used by GUI-based tools.

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 (e.g., to copy, rename, project attributes or 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` and compare the development efforts for visual and code-based programming, the full paper about `pygrametl` [24] presented an evaluation where the running example was implemented in both `pygrametl` and the graphical ETL tool Pentaho Data Integration (PDI) [12], a popular Java-based open source ETL tool. 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 any evaluation/performance results without the consent of the provider, so this was not possible. In this section, we present the findings of the evaluation. `pygrametl`, the case ETL program, and the data generator are publicly available from pygrametl.org.

9.1 Development Time

It is obviously difficult to make a comparison of two such tools, and a full-scale test would require several teams of fully trained developers, which is beyond our resources. We obviously know `pygrametl` well, but also have solid experience with PDI from earlier projects. Each tool was used 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 program was very easy to develop. It took a little less than an hour in the first use, and 24 minutes in the second. The program consists of $\sim$140 short lines, e.g., only one argument per line when creating `Dimension` objects. This strongly supports that it is easy to develop ETL programs using `pygrametl`. The main method of the ETL is shown below.

```
def main():
    for row in inputdata:
        extractdomaininfo(row)
        extractserverinfo(row)
        row['size'] = pygrametl.getint(row['size'])
        # 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 to extract domain, top-level domain, and server name. Note that the page dimension is an SCD, where `scdensure` is a very easy way to fill a both snowflaked and slowly changing dimension. The date dimension is filled using a `rowexpander` for the `datedim` object to (on demand) calculate the attribute values so 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 PDI-based solution took us a little more than two hours in the first use, and 28 minutes in the second. The flow is shown in Figure 3. We emulate

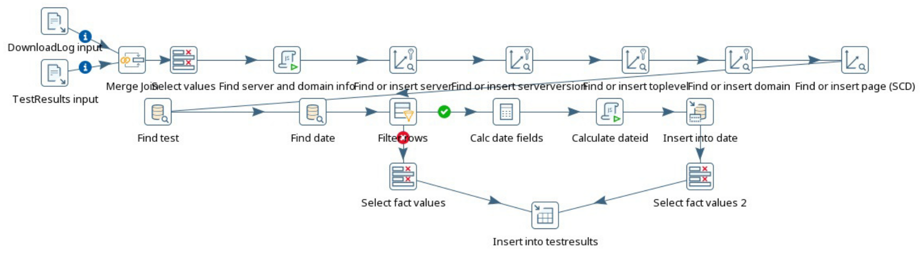


Fig. 3. Data flow in PDI-based solution.

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.

To summarize, `pygrametl` was faster to use than PDI. The first solution was much faster to create in `pygrametl` and we believe that the strategy is far simpler to work out in `pygrametl` (compare the shown main method and Figure 3). And although trained PDI users also may be able to generate a first solution in PDI quickly, we still believe that the `pygrametl` approach is simple and easy to use which is also seen from the second uses of the tools where it was as fast (actually a little faster) to type code as to click around in the GUI to (re)create the prepared solutions.

9.2 Performance

The original full paper [24] also presented performance results for both PDI and `pygrametl` on the running example. In the current chapter, we provide new results for the same example, but with the newer versions of both PDI (version 7.1) and `pygrametl` (version 2.5) and executed on newer hardware.

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 $\sim$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] virtual machine with three virtual processors, and 16GB of RAM (the CPUs were never completely used during the experiments and the amount of RAM was big enough to allow both `pygrametl` and PDI to cache all dimension data). The virtual machine ran openSUSE Leap 42.2 Linux, pygrametl 2.5 on Python 3.6, PDI 7.1 on OpenJDK 8, and PostgreSQL 9.4. The virtual machine was running under VirtualBox 5.1 on Windows 10 on a host machine with 32GB of RAM, SSD disk, and a 2.70GHz Intel i7 CPU with 4 cores and hyperthreading.

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)`.

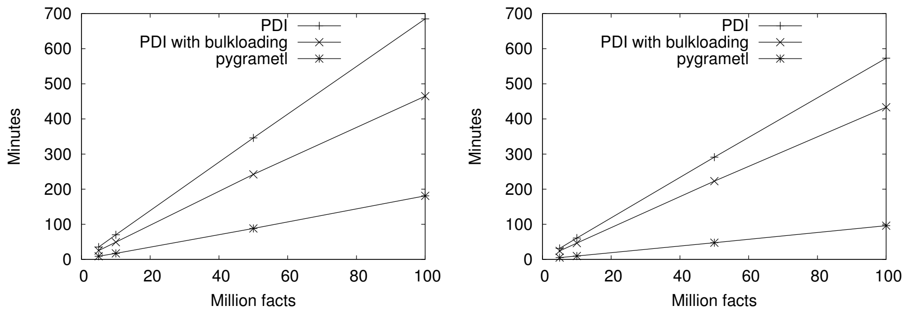
PDI was tested in two modes. One with a single connection to the DW such that the ETL is transactionally *safe* and one which uses a special component for bulkloading the facts into PostgreSQL. This special component makes its own connection to the DW. This makes the load faster but transactionally *unsafe* as a crash can leave the DW loaded with inconsistent data. The `pygrametl`-based solution uses bulkloading of facts (by means of `BulkFactTable`) but is always running in a *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 maximal Java heap size was set to 12GB.

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 amounts of time grow linearly for both PDI and `pygrametl`.

`pygrametl` is significantly faster than PDI in this experiment. When loading 100 million facts, the `pygrametl`-based solution handles 9208 facts/sec. PDI with a single connection handles 2433 and PDI with two connections handles 3584 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` also uses much less CPU time than PDI. For example, when loading the data set with 100 million facts, `pygrametl`'s CPU utilization is 53%. PDI's CPU utilization is 83% with one connection and 93% with two. It is clearly seen that it in terms of resource consumption, it is beneficial to code a specialized light-weight program instead of using a general feature-rich but heavy-weight ETL application.

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



(a) Elapsed time (b) CPU time

Fig. 4. Performance results.

10 Parallelism

ETL flows often have to handle large data volumes and parallelization is a natural way to achieve good performance. `pygrametl` has support for both task parallelism and data parallism [25]. The support was made to keep the simplicity of `pygrametl` programs For example, extraction and transformation of data can happen in another process using a **`ProcessSource`**:

```
rawdata = CSVSource(open('myfile.csv', 'r'))
transformeddata = TransformingSource(rawdata, transformation1, transformation2)
inputdata = ProcessSource(transformeddata)
```

The ProcessSource will here spawn another process in which the rows of `transformeddata` are found and then delivered to the `inputdata` object in the main process. In a similar way, `Dimension` instances can also do their work in another process, by means of **`DecoupledDimension`** which has an interface identical to `Dimension`, but just pushes all work to a `Dimension` instance in another process. This other instance is given to `DecoupledDimension` when it is created as exemplified below:

```
pagedim = DecoupledDimension(SlowlyChangingDimension(name='page', ...))
```

Work on a `FactTable` instance can also happen in another process by means of **`DecoupledFactTable`**. Decoupled instances can also be defined to *consume* data from each other such that, e.g., lookup operations don't become blocking but rather return a placeholder value that a consuming class later will get without involvement from the user (or main process). For details, see [25].

```
facttbl = DecoupledFactTable(BulkFactTable(name='testresults', ...),
                             consumes=pagedim,
                             returnvalues=False)
```

Finally, `pygrametl` has support for making user-defined functions run in parallel with other tasks by means of using a **`splitpoint`** annotation.

```
@splitpoint(instances=2)
def myfunction(*args)
    # Do some (possibly expensive) transformations here
```

In [25], it is concluded that the simple constructs for parallelism give good improvements and that "a small increase in used CPU time gives a large increase in performance. The available CPUs are used more intensively, but for a shorter amount of (wall-clock) time". In the `pygrametl` implementation, new processes are spawned when `pygrametl` is executed on CPython, the reference implementation for Python. The reason is that Python threads cannot execute Python bytecode in parallel on CPython due to the global interpreter lock (GIL). Use of threads on CPython can thus result in poor performance. When `pygrametl` is running on Jython, the implementation of Python in Java, threads are used instead since their performance is good on the JVM. It is, however, possible to tune the performance of both processes and threads by setting defining sizes for *batches* of row and the *queues* they are placed in when transferred to another proces/thread. We have attempted to pick reasonable default values, but have also experienced that other values may increase or decrease the performance significantly. Unfortunately, it is not easy to pick good values as sizes that work well on one machine can be unappropriate for another machine. Automatic and dynamic selection of good values for these sizes is thus an interesting future task.

The functionality described above enables parallelism on a single computer. However, it becomes infeasible to scale up to very large amounts of data and instead it is necessary to scale out. This is exactly what the MapReduce [2] framework does. It is thus interesting to apply MapReduce to ETL programming. However, while MapReduce offers high flexibility and scalability, it is a generic programming model and has no native support of ETL-specific constructs such as star schemas, snowflake schemas, and slowly changing dimensions. Implementing a parallel ETL procedure on MapReuce is therefore complex, costly, and error-prone and leads to low programmer productivity. As a remedy, Liu et al. [9] presented ETLMR which is a dimensional ETL framework for MapReduce with direct support for high-level ETL constructs. The general idea is that ETLMR leverages the functionality of MapReduce, but also hides the complexity. To achieve this, the user only specifies transformations and declarations of sources and targets which only requires few lines of code. The implementation of ETLMR is based on `pygrametl` but some parts have been extended or modified to support MapReduce.

To exploit the strenghts of MapReduce, the user has a little less fredom with ETLMR than with `pygrametl`. With ETLMR, an ETL flow always consists of dimension processing first followed by fact processing in another MapReduce job. In a file, config.py, the user has to declare sources and targets as well as a mapping called dims which tells which attributes are needed for each dimension and which transformations (i.e., Python functions) to apply to dimension data first. An example for our running example is shown below.

```
from odottables import *  # Different dimension processing schemes are supported
fileurls = ['dfs://.../TestResults0.csv' ,'dfs://.../TestResults1.csv', ...]

datedim = CachedDimension(...)  # Similar to declarations in pygrametl
pagedim = SlowlyChangingDimension(...)
testdim = CachedDimension(...)

dims = {pagedim:{'srcfields':('url', 'serverversion', 'domain',
                              'size', 'lastmoddate'),
                 'rowhandlers':(UDF_extractdomain, UDF_extractserver)},
        datedim:...,
        testdim:...
}
```

There are different dimension processing schemes available in ETLMR. The first one is called *One Dimension, One Task (ODOT)*. In this scheme, mappers will project attributes needed for the different dimensions and emit pairs of the form (dimension name, list of attribute values). One reducer will then process *all* data for one dimension and apply user-defined transformations, do key generation, and fill/update the dimension table. Another scheme is called *One Dimenion, All Tasks (ODAT)*. In ODAT, mappers will also project attributes, but here they for each input row emit one key/value pair with data for all dimensions, i.e., a pair of the form (rownumber, [dimension1:{...}, dimension2:{...}, ...]). These pairs are then distributed to the reducers in a round-robin fashion and one reducer thus processes data for all dimensions and one dimension is processed by all reducers. This can, however, lead to inconsistencies and therefore a final step is needed to clean up such inconsistencies [9]. A hybrid of ODAT and ODOT is also possible. In this hybrid scheme, a data-intensive dimension (such as pagedim) can be partitioned based on business key (url) and processed by all tasks which is ODAT-like while the remaining dimensions are processed in ODOT-style.

11 Case Study

In this section, we describe how and why a concrete company uses programmatic ETL and `pygrametl` for its DW solutions.

The company FlexDanmark[4] handles demand-responsive transport in Denmark. For example, elderly people are picked up at home and driven to a hospital for examinations and school children living in areas without access to public transportation are driven from home to school. The yearly revenue is around 120 million US dollars. To organize the transportation and plan the different tours effectively, FlexDanmark makes routing based on detailed speed maps built from GPS logs from more than 5,000 vehicles. Typically, 2 million GPS coordinates are delivered to FlexDanmark every night. FlexDanmark has a DW where the cleaned GPS data is represented and integrated with other data such as weather data (driving speeds are related to the weather and are, e.g., lower when there is snow). The ETL procedure is implemented in Python. Among other things, the ETL procedure has to transform between different coordinate systems, do map matching to roads (GPS coordinates can be quite imprecise), do spatial matching to the closest weather station, do spatial matching to municipalities and zip code areas, and finally load the DW. The latter is done by means of `pygrametl`.

The trips handled by FlexDanmark are paid by or subsidized by public funds. To be able to analyze how money is spent, another DW at FlexDanmark therefore holds data about payments for the trips, the taxi companies providing the vehicles, customers, etc. This DW integrates data from many different source systems and has some interesting challenges since payment details (i.e., facts in the DW) about already processed trips can be updated, but the history of these updates must by tracked in a similar way to how dimension changes are tracked for a type-2 slowly changing dimension. Further, new

[4] `http://flexdanmark.dk`. One of the authors (Ove Andersen) is employed by FlexDanmark.

sources and dimensions are sometimes added. The ETL procedure for the DW is also implemented in Python, but FlexDanmark has created a framework that by means of *templates can generate Python code* incl. `pygrametl` objects based on metadata parameters. Thus, FlexDanmark can easily and efficiently generate ETL code that handles parallelism, versioning of facts, etc. when new sources and/or dimensions are added.

FlexDanmark's reasons for using code-based, programmatic ETL are manifold. FlexDanmark's DWs are rather advanced since they handle GPS data and versioned facts, respectively. To implement ETL proedures for these things in traditional GUI-based tools was found to be hard. FlexDanmark did in fact try to implement the map matching in a widely used commercial ETL tool, but found it hard to accomplish this task. In contrast, it was found to be quite easy to do programmatically in Python where existing libraries easily could be used and also easily replaced with others when needed. Programmatic ETL does thus give FlexDanmark bigger flexibility. Yet another aspect is pricing since FlexDanmark is publicly funded. Here programmatic ETL building on free, open source software such as Python and `pygrametl` is desirable. It was also considered to use a free, open source GUI-based ETL tool, but after comparing a few programmatic solutions with ther counterparts in the GUI-based tool, it was found to be faster and more flexible to code the ETL procedures.

In most cases where `pygrametl` is used, we are not aware for what since users are not obliged to register in any way. A few `pygrametl` users have, however, told us in private communication how they use `pygrametl`. We thus know that `pygrametl` is used in production systems from a wide variety of domains including health, advertising, real estate, public administration, and sales.

Based on the feedback we have received, we have so far not been able to extract guidelines or principles for how best to design programmatic ETL processes. The programming involved can vary from very little for a small proof of concept to a significant amount for a multi-source advanced DW. The principles to apply should thus probably be those already applied in the organization. Reusing existing and known development principles also reduces the time required to learn `pygrametl` as users only have to get to know a new Python library, but not new development principles. `pygrametl` does not make any strict assumptions about how the program should be organized (for example, it is *not* required that dimension data is processed before fact data). `pygrametl` is designed as a library where the user can make objects for the dimension tables or fact tables she wishes to operate on. `pygrametl` will then do all insertions into and updates of the underlying database while the user can focus on and structure the surrounding logic as she pleases.

12 Experiences from Open-Sourcing **`pygrametl`**

In this section, we describe our experiences with open-sourcing `pygrametl`.

When the first paper about `pygrametl` was published, we also made the source code available for download from our department's web page. From logs, we could see that there were some downloads and we also received some questions and comments, but not too many. Later, the source code was moved to Google Code and the received attention increased. When Google Code was taken out of service, the code was moved to GitHub[5] where we got a lot more attention. The lesson we learned from this is that it is very important to publish source code at a well-known place where people are used to look for source code. In fact, before we went to GitHub, others had already created unofficial and unmaintained repositories with the `pygrametl` code outside our control. Anyone is of course free to take, modify, and use the code as they please, but we prefer to be in control of the repository where people get the code to ensure availability of new releases. Along the same lines, we also learned that it is important to make it easy for users to install the library. For a Python library as `pygrametl`, it thus important to be available on the Python Package Index (PyPI[6]). Again we experienced that unofficial and unmaintained copies were created before we published our official version. With `pygrametl` on PyPI, installation of `pygrametl` is as simple as using the command `pip install pygrametl`.

Another lesson learned – although not very suprising – is that good documentation is needed. By making a Beginner's Guide and examples available online [7], we have reduced the number of (repeated) questions to answer via mail, but also made it easy to get started with `pygrametl`. It is also important to describe early what the tool is intended to do. For example, we have now and then received questions about how to do general data movement from one platform to another, but that is not what `pygrametl` is inteded for. Instead, the focus for `pygrametl` is to do ETL for *dimensional* DWs.

Finally, we have also received very good help from users. They have found – and improved – performance bottlenecks as well as generalized and improved functionality of `pygrametl`.

`pygrametl` is published under a BSD license. We have chosen this license since it is a very permissive license. The BSD license allows proprietary use of the licensed code and has no requirements about making derivative work publicly available. In principle, there is thus a risk that users could improve the `pygrametl` source code without sharing the improvements with us. However, we prefer to give users freedom in deciding how to use the `pygrametl` code and, as described above, we do get suggestions for code improvements from users.

13 Conclusion and Future Work

We have presented `pygrametl` which is a programming framework for ETL programming. We challenge the conviction that ETL development is always best done in a graphical tool. We propose 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.

[5] `http://chrthomsen.github.io/pygrametl/`
[6] `http://pypi.python.org/pypi`
[7] `http://chrthomsen.github.io/pygrametl/doc/index.html`

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.

`pygrametl` is available as open source and is used in proofs of concepts as well as production systems from a variety of different domains. We have learned the importance of publishing code at well-known places such as GitHub and the joy of users contributing improvements. When code is added or changed, we try hard to ensure that existing code does not break. For this, a future focus area is to automate testing much more than today. For future major releases, it can also be considered to introduce a new API with fewer classes, but the same or more functionality. The current class hierarchy to some degree reflects that new functionality has been added along the way when someone needed it. The way to load rows (plain SQL INSERTs, batched INSERTs, or by bulkloading) is now defined by the individual classes for tables. A more general approach could be by composition of loader classes into the classes for handling dimensions and fact tables. It would also be interesting to investigate how to allow generation of specialized code for the task at hand by using templating where the user can select features to enable such as bulkloading. This could potentially give big performance advantages. A strength of `pygrametl` is the easy integration with other Python projects. More intergration with relevant projects such as data sources for Pandas [8] would also be beneficial.

References

1. M. A. Beyer, E. Thoo, M. Y. Selvage, and E. Zaidi: "Gartner Magic Quadrant for Data Integration Tools", 2017
2. J. Dean and S. Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters". In *Proc. OSDI*, pp. 137–150, 2004.
3. Django. `djangoproject.com/` as of 2017-10-13.
4. H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Text-based Modeling. In *Proc. of ATEM*, 2007
5. IBM InfoSphere DataStage. `https://www.ibm.com/ms-en/marketplace/datastage` as of 2017-10-13
6. Informatica. `informatica.com` as of 2010-10-13.
7. C. S. Jensen, T. B. Pedersen, and C. Thomsen. *Multidimensional Databases and Data Warehousing*. Morgan & Claypool, 2010.
8. R. Kimball and M. Ross. *The Data Warehouse Toolkit*, 2nd Edition. Wiley, 2002.
9. X. Liu, C. Thomsen, and T. B. Pedersen: "ETLMR: A Highly Scalable Dimensional ETL Framework based on MapReduce". In *Proc. DaWaK*, pp. 96–111, 2011.
10. Microsoft SQL Server Integration Services. `https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services` as of 2017-10-13.
11. Oracle Data Integrator. `http://www.oracle.com/technetwork/middleware/data-integrator/overview/index.html` as of 2017-10-13.

[8] `http://pandas.pydata.org/`

12. Pentaho Data Integration - Kettle. `http://kettle.pentaho.org` as of 2017-10-13.
13. M. Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Comm. ACM* 38(6):33-44, 1995.
14. PostgreSQL. `postgresql.org` as of 2017-10-13.
15. Psycopg. `http://initd.org/psycopg/` as of 2017-10-13.
16. Python. `python.org` as of 2017-10-13.
17. Ruby on Rails. `rubyonrails.org/` as of 2017-10-13.
18. SAP Data Services. `https://www.sap.com/products/data-services.html` as of 2017-10-13.
19. Scriptella. `scriptella.org` as of 2017-10-13.
20. 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.
21. C. Thomsen and T.B. Pedersen. Building a Web Warehouse for Accessibility Data. In *Proc. of DOLAP*, 2006.
22. C. Thomsen and T.B. Pedersen. A Survey of Open Source Tools for Business Intelligence. *IJDWM* 5(3):56-75, 2009.
23. C. Thomsen and T. B. Pedersen: "pygrametl: A Powerful Programming Framework for Extract–Transform–Load Programmers". In *Proc. DOLAP*, pp. 49–56, 2009.
24. C. Thomsen and T.B. Pedersen. *pygrametl: A Powerful Programming Framework for Extract–Transform–Load Programmers*. DBTR-25, Aalborg University. `www.cs.aau.dk/DBTR`, 2009.
25. C. Thomsen and T. B. Pedersen: "Easy and Effective Parallel Programmable ETL". In *Proc. DOLAP*, pp. 37–44, 2011.
26. J. Trujillo and S. Lujàn-Mora. A UML Based Approach for Modeling ETL Processes in Data Warehouses. In *Proc. of ER2003*, pp. 307–320, 2003.
27. A. Vaisman and E. Zimanyi. *Data Warehouse Systems: Design and Implementation*. Springer, 2014.
28. P. Vassiliadis. A Survey of Extract–Transform–Load Technology. *IJDWM* 5(3):1-27, 2009.

Appendix A Data Warehouse Concepts

This appendix offers a very short introduction to concepts and terms used in the chapter. More details and explanations can be found in the literature [7,8,27]. In a *data warehouse* (DW), data from an organization's different operational systems is stored in a way that supports analysis (rather than the daily operations which are supported by the operational systems). An *Extract-Transform-Load* (ETL) process extracts data from the source systems, transforms the data (to make it fit into the DW and to cleanse it), and loads it into the DW. Data is divided into *facts* and *dimensions*. Facts represent the subjects of the desired analyses (e.g., sales) and have numerical *measures* (e.g., sales amount). Dimensions provide context and describe facts (Product, Store, and Time are, for example, relevant dimensions for sales). Dimensions are thus used for selection of data and grouping of data in analyses. Dimensions have *hierarchies* with levels (a Time dimension can, for example, have the hierarchy Day $\rightarrow$ Month $\rightarrow$ Quarter $\rightarrow$ Year. Each of the levels can also have a number of attributes.

When using a relational database to represent a DW, one can choose between two approaches for the schema design. In a *snowflake schema*, each level in a dimension is represented by a table and the tables have foreign keys to the following levels. The dimension tables are thus normalized. In a *star schema* there is only one table for each dimension. This table thus represents all levels and is denormalized. In both star schemas and snowflake schemas, the facts are represented by a *fact table* which has a foreign key for each dimension and a column for each measure. In a star schema, the foreign keys reference the dimension tables while they reference the tables for the lowest levels of the dimensions in a snowflake schema. The keys used in a dimension table should be integers not carrying any special meaning. Such keys are called *surrogate keys*.

Changes may happen in the represented world. It is thus necessary to be able to represent changes in dimensions. A dimension where changes are represented is called a *slowly changing dimension* (SCD). There are a number of different techniques for SCDs [8]. Here we will consider two of the most commonly used. For *type 1 SCDs*, changes are simple represented by overwriting old values in the dimension tables. If, for example, the size of a shop changes, the size attribute is updated. This can be problematic as old facts (e.g., facts about sales from the shop when it had the previous size) now refer to the updated dimension member such that history is not correctly represented. This problem is avoided with a *type 2 SCD* where a new *version* of the dimension member is created when there is a change. In other words, for type 2 SCDs, changes result in new rows in the dimension tables. In a type 2 SCD, there are often attributes called something like ValidFrom, ValidTo, MostCurrentVersion, and VersionNumber to provide information about the represented versions.