pygrametl: A Powerful Programming Framework for Easy Creation and Testing of ETL Flows

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

Published in:
Transactions on Large-Scale Data- and Knowledge-Centered Systems XLVIII

DOI (link to publication from Publisher):
10.1007/978-3-662-63519-3_3

Creative Commons License
CC BY 4.0

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Jensen, S. K., Thomsen, C., Pedersen, T. B., & Andersen, O. (2021). pygrametl: A Powerful Programming Framework for Easy Creation and Testing of ETL Flows. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XLVIII: Special Issue In Memory of Univ. Prof. Dr. Roland Wagner* (Vol. XLVIII, pp. 45-84). Springer. https://doi.org/10.1007/978-3-662-63519-3_3

pygrametl: A Powerful Programming Framework for Easy Creation and Testing of ETL Flows

Søren Kejser Jensen[1](✉), Christian Thomsen[1], Torben Bach Pedersen[1], and Ove Andersen[2]

[1] Department of Computer Science, Aalborg University, Aalborg, Denmark
{skj,chr,tbp}@cs.aau.dk
[2] Danish Geodata Agency, Nørresundby, Denmark
ovand@gst.dk

Abstract. Extract-Transform-Load (ETL) flows are used to extract data, transform it, and load it into data warehouses (DWs). The dominating ETL tools use graphical user interfaces (GUIs) where users must manually place steps/components on a canvas and manually connect them using lines. This provides an easy to understand overview of the ETL flow but can also be rather tedious and require much trivial work for simple things. We, therefore, challenge this approach and propose to develop ETL flows by writing code. To make the programming easy, we proposed the Python-based ETL framework `pygrametl` in 2009. We have extended `pygrametl` significantly since the original release, and in this paper, we present an up-to-date overview of the framework. `pygrametl` offers commonly used functionality for programmatic ETL development and enables the user to efficiently create effective ETL flows with the full power of programming. Each dimension is represented by a dimension object that manages the underlying table or tables in the case of a snowflaked dimension. Thus, filling a slowly changing or snowflaked dimension only requires a single method call per row as `pygrametl` performs all of the required lookups, insertions, and assignment of surrogate keys. Similarly to dimensions, fact tables are each represented by a fact table object. Our latest addition to `pygrametl`, Drawn Table Testing (DTT), simplifies testing ETL flows by making it easy to define both preconditions (i.e., the state of the database before the ETL flow is run) and postconditions (i.e., the expected state after the ETL flow has run) into a test. DTT can also be used to test ETL flows created in other ETL tools. `pygrametl` also provides a set of commonly used functions for transforming rows, classes that help users parallelize their ETL flows using simple abstractions, and editor support for working with DTT. We present an evaluation that shows that `pygrametl` provides high programmer productivity and that the created ETL flows have good run-time performance. Last, we present a case study from a company using `pygrametl` in production and consider some of the lessons we learned during the development of `pygrametl` as an open source framework.

O. Andersen—The work was done while employed at Aalborg University and FlexDanmark.
The original version of this chapter was revised: affiliation, email address of the last author and ref. 17 have been updated. The correction to this chapter is available at https://doi.org/10.1007/978-3-662-63519-3_9

A. Hameurlain and A Min Tjoa (Eds.): TLDKS XLVIII, LNCS 12670, pp. 45–84, 2021.
https://doi.org/10.1007/978-3-662-63519-3_3

1 Introduction

The Extract-Transform-Load (ETL) flow is a crucial part of a data warehouse (DW) project. The task of an ETL flow is to extract data from possibly heterogeneous data sources, perform transformations (e.g., conversions and cleaning of data), and finally load the transformed data into a DW. It is well-known in the DW community that it is both very time-consuming and difficult to get the ETL right due to its high complexity [24]. 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 support users in implementing ETL flows [4,55]. The leading ETL tools provide graphical user interfaces (GUIs) in which the users define the flow of data visually. While these are easy to use and inherently provide an overview of the ETL flow, they also have disadvantages. For example, users might require a component that is not provided by the graphical ETL tool. Instead, users must create solutions based on (often complicated) combinations of the provided components or by integrating custom-coded components into the ETL flow. Both are very time-consuming solutions.

In an ETL project, non-technical staff is often involved as advisors, decision makers, etc. but the core development is (in our experience) done by skilled ETL specialists. As trained specialists often can use textual interfaces efficiently while non-specialists use GUIs, it is attractive to consider alternatives to graphical ETL tools. In relation to this, one can recall the high expectations of 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 would then generate the code. Needless to say, these expectations were not fulfilled. It can be argued that forcing all ETL development into GUIs is a step back to the CASE idea.

We acknowledge that graphical ETL tools can be useful, but we also claim that for many ETL projects, a code-based solution is the right choice. Often it is simply much faster to express the desired operations in a few lines of code instead of placing components on a canvas and setting properties in dialog boxes. However, many parts of code-based ETL programs are redundant if each is written from scratch. To remedy this, a framework with common functionality is needed.

In this paper, we present an overview of the latest version of ***pygrametl*** (version 2.7), an open source Python-based framework for ETL programmers we first proposed in 2009 [56]. The framework provides common functionality for ETL development and while it is easy to get an overview of and start using, it is still very powerful. `pygrametl` offers a novel approach to ETL programming by providing a framework that abstracts over the DW tables while still allowing the user to use the full power of Python. For example, it is very easy to create (relational or non-relational) data sources as they are just iterable objects that produce *rows*. Thus, using a simple loop users can insert data into dimension and fact tables while only iterating over the source data once. Dimensions and fact tables are each represented by a *single object* which manages the underlying

DW tables. For example, use of snowflaked dimensions is very easy as the user only operates on *one* dimension object for the entire snowflake while `pygrametl` automatically manages all of the tables in the snowflake. Thus, filling a snowflaked dimension only requires a single method call per row. Users who prefer the inherent structure provided by graphical ETL tools can also optionally create their ETL flows using connected step objects.

For testing, `pygrametl` provides DTT that makes it simple to define a self-contained test that sets the state of a DW before a test is run (the precondition) and verifies it has a certain state afterward (the postcondition). DTT can, e.g., be used to create unit tests for each component of an ETL flow during development or to create integration tests that ensure the components work together. Exploiting parallelism to improve the performance of an ETL flow is also simple with `pygrametl`, e.g., transformations implemented as functions can be parallelized with only one line of code. In general, `pygrametl` utilizes functional and object-oriented programming to make ETL development easy. `pygrametl` is thus similar to other special-purpose frameworks that provide commonly used functionality like, e.g., the web frameworks Django [8] and Ruby on Rails [45] where development is done in Python and Ruby code, respectively. Our evaluation in Sect. 10 shows that `pygrametl` provides high programmer productivity and that the programmatically created ETL flows have good run-time performance when compared to a leading open source graphical ETL tool.

The paper is an extended version of [52] and is structured as follows: Sect. 2 presents an ETL scenario we use as a running example. Section 3 gives an overview of `pygrametl`. Sections 4–8 present the functionality and classes provided by `pygrametl` to support *data sources*, *dimensions*, *fact tables*, *flows*, and *testing*, respectively. Section 9 provides a short overview of the supporting functionality provided by `pygrametl` including helper functions, support for parallelism, and editor support. Section 10 evaluates `pygrametl` on the running example. Section 11 presents a case-study of a company using `pygrametl`. Section 12 documents our experience with publishing `pygrametl` as open source. Section 13 presents related work. Section 14 concludes and points to future work.

2 Example Scenario

We now 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 [53] but it has been simplified here for the sake of brevity.

In the system, there is a web crawler that downloads web pages. Each downloaded page is stored in a local file. The crawler stores data about the downloaded files in a log which is a tab-separated file with the fields shown in Table 1(a). Afterward, 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 the test outputs the number of errors detected. The results of the tests are written to a tab-separated file with the fields shown in Table 1(b).

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

After all tests are performed, the data from the two files are loaded into a DW by an ETL flow. The schema of the DW is shown in Fig. 1. The schema has three dimensions: The test dimension holds information about each of the tests that are applied. This dimension is static and prefilled. The date dimension is filled by the ETL on-demand. The page dimension is *snowflaked* and spans several tables. It holds information about the downloaded 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* (SCD) [23] where information about different *versions* of a given 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 detail here. There is one fact table that 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

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 [23] managed by RDBMSs. Focusing on RDBMSs as the target for `pygrametl` keeps the design simple as it allows us to create good solutions that are specialized for this domain instead of thinking in very general *data integration terms*. The data sources, however, do *not* have to be relational.

When using `pygrametl`, the programmer writes code that controls the flow, the extraction (the E in ETL) from source systems, the transformations (the T

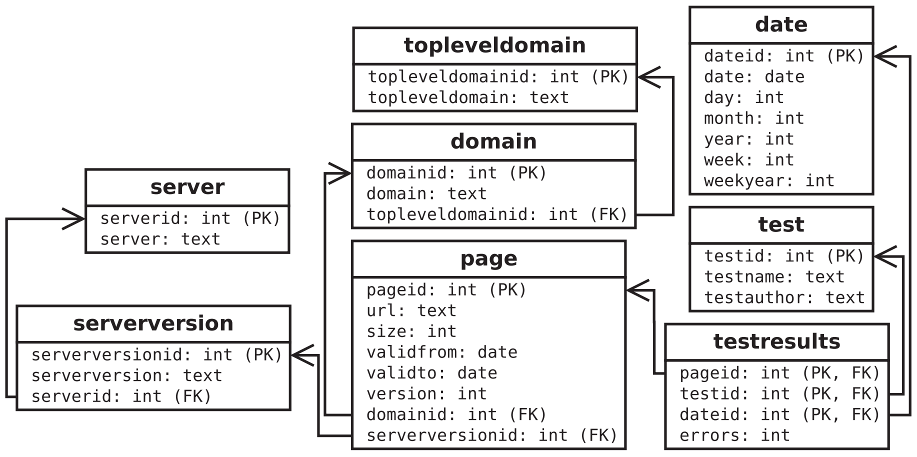


Fig. 1. The schema for the running example.

in ETL) of the source data, and the loading (the L in ETL) of the transformed data. For the flow control, extraction, and load, `pygrametl` offers components that support the user and makes it easy to create additional components. For the transformations, the user benefits heavily from being able to use Python.

Loading data into a DW is particularly easy with `pygrametl`. The general idea is that the user creates *objects* for each *fact table* and *dimension* (different kinds are supported) in the DW. An object representing a dimension offers convenient methods like `insert`, `lookup`, etc. that hide details of key assignment, SQL insertion, caching, etc. In particular, it should be noted that a snowflaked dimension is also treated in this way such that a single object can represent the entire dimension although the data is inserted into several tables in the database.

The dimension object's methods take *rows* as arguments. A row in `pygrametl` is a mapping from names to values. Based on our experiences with other tools, we found it important that `pygrametl` does not try to validate that all rows given to a dimension object have the same attributes or the same attribute types. If the user wants such checks, they can write code for that. It is then, e.g., possible to have additional attributes in a row or on purpose leave out certain attributes. `pygrametl` only raises an error if attributes required for an operation are missing. For example, all values to be inserted into a 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 callback of user-defined functions to compute the missing values. So, while some tools enforce uniformity of rows, `pygrametl` is designed to make it easy for the users to do what they want instead of what the tool *thinks* they want.

While `pygrametl` is written in Python [44] other languages could have been used. We chose Python due to its focus on programmer productivity and its large selection of libraries. Also, Python is dynamically typed (the type of a variable is checked at run-time) and strongly typed (a variable holding an integer cannot be used as a string). Consider, e.g., the Python function in Listing 1.

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

Listing 1. Function that casts `value` to a float and returns `default` otherwise.

This function converts its input to a `float` or, if the conversion fails, to a value which defaults to `None`, Python's `null` value. Note that no static types are specified. The function can be called with different types as shown in Listing 2.

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

Listing 2. Using `getFloat` with and without a default value.

After executing the code, the value of `f1`, `f2`, and `f3` will 10.0 while the value of `f4` will be the string `Not a float!`. Thus, the expression `f1 + f2` will succeed, while `f3 + f4` will fail as a float and a string cannot be added together.

Python is predominantly object-oriented but to some degree also supports functional programming, e.g., by allowing use of functions or lambda expressions as arguments. This makes it very easy to customize behavior. `pygrametl`, for example, exploits this to support compute of missing values on-demand (see Sect. 5). Using Python's support for default arguments, `pygrametl` also provides reasonable defaults when possible to spare the user for unnecessary typing.

4 Data Source Support

`pygrametl` supports a large selection of data sources and makes it easy to add more. As explained in Sect. 3, data is moved around in *rows* in `pygrametl`. Instead of implementing a row class, we use Python's built-in dictionaries (`dict`) as it provides efficient mappings between keys (i.e., attribute names) and values (i.e., attribute values). The only requirement of data sources in `pygrametl` is that they are iterable (i.e., they must define the `__iter__` method) and produce `dict`s. Thus, it does not require a lot of programming to create new data sources. Of course, the complexity of the code that extracts the data heavily depends on the source format. `pygrametl` includes the following data sources.

`SQLSource` is a data source that returns 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 created.

`CSVSource` is a data source that returns the lines of a delimiter separated file as rows. This class is implemented in `pygrametl` as a reference to the class `csv.DictReader` in Python's standard library. In the running example, 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 shown in Listing 3.

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

Listing 3. Reading a CSV file using the `CSVSource` data source.

`TypedCSVSource` functions like `CSVSource` but can also performs type casting of its input. In Listing 4 values from the field `size` are cast to `int`s.

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

Listing 4. Reading a CSV file using the `TypedCSVSource` data source.

`PandasSource` wraps a Pandas DataFrame [33] so it can be used as a data source as shown in Listing 5. Thus, Pandas can be used to extract and transform data before `pygrametl` loads it.

```
df = pandas.read_parquet("TestResults.parquet")
df = df.dropna()
testresults = PandasSource(df)
```

Listing 5. Using a Pandas DataFrame as a data source through `PandasSource`.

The above data sources take external data as input, while the following data sources take data sources as input to combine them or perform transformations.

`MappingSource` generalizes the cast concept from `TypedCSVSource`. For each row from a data source it executes arbitrary functions on its attribute values.

Both `TypedCSVSource` and `MappingSource` apply transformations to attribute values. `TransformingSource` in contrast applies transformations to rows, and can thus, e.g., add or remove attributes. For these three data sources, Python's support for functional programming is used as functions are passed as arguments.

`MergeJoiningSource` is a data source that equijoins rows from two other data sources. Thus, it must be given two data sources (which must deliver rows in sorted order) and the attributes to join on. Both of the data sources in the running example have the field `localfile` so an equijoin can be done as shown in Listing 6 where `testresults` and `downloadlog` are `CSVSource`s.

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

Listing 6. Joining the rows from two data source using `MergeJoiningSource`.

`HashJoiningSource` is a data source that equijoins rows from two other data sources using a hash map. Thus, the input data sources need not be sorted.

`UnionSource` and `RoundRobinSource` union rows from data sources together. `UnionSource` returns all rows from a data source before using the next, while `RoundRobinSource` switches after a user-defined number of rows are taken. A `DynamicForEachSource` automatically creates multiple data sources but presents them as one data source. This is, e.g., useful for a directory containing many CSV files. The user must provide a function that when called with an argument returns a data source and a sequence of arguments for it as shown in Listing 7.

```
srcs = DynamicForEachSource([... sequence of the names of the files ...],
                            lambda f: CSVSource(open(f, 'r')))
for row in srcs: # will iterate over all rows from all the files;
   ...           # the user can transform and load each row here
```

Listing 7. Reading a sequence of CSV files using `DynamicForEachSource`.

Last, `CrossTabbingSource` can pivot data from another data source, while data from another data source can be filtered using `FilteringSource`.

5 Dimension Support

The classes `pygrametl` uses for representing dimensions in a DW are shown in Fig. 2. Only public methods and their required arguments are shown. Note that `SnowflakedDimension` does not inherit from `Dimension` but offers the same interface and can be used as if it were a `Dimension` due to Python's duck typing.

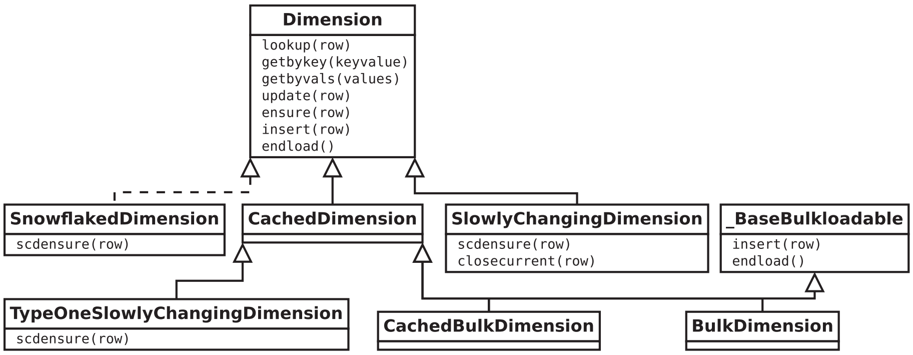


Fig. 2. Class hierarchy for the dimension supporting classes.

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 name of the key column (we assume that a dimension has a non-composite key), and a list of attributes (the underlying table may have more attributes but `pygrametl` will then not manage them) must be given. By default, all of the none-key attributes are used for looking up the key value. To only use a subset of the attributes, a list of *lookup attributes* (`lookupatts`) can be given. In the running example, the test dimension has a surrogate key testid but in the CSV files, each is identified by testname. Thus, the value of the key must be found based on the test name and not all of test's attributes.

When the dimension object is given a row to insert into the table (explained below), the row need not contain a value for the dimension's key. If the key is not present, 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 non-existing dimension members can also be set (`defaultidvalue`). 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 should still 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 flow. If data from the 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 foreign key 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 is shown in Listing 8.

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

Listing 8. Representing the test dimension using `Dimension`.

Finally, the user can provide a function as the argument `rowexpander`. With such a function, it possible to add required fields on-demand to a row before it is inserted into the dimension (explained in detail below).

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. In the running example, the rows from the source file TestResults.csv contain 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 $n = \{\text{'testname':'test'}\}$ can then be set up such that when `py-grametl` 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 that 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, 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 computed from the date string in the source data. However, it should only be computed once for each 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 a date.

`CachedDimension` has the same interface and semantics as `Dimension`. However, it internally caches dimension members in memory 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 when added.

When an instance of `CachedDimension` is created, it is possible to provide the same arguments as for `Dimension`. Further, optional arguments can be used to set the size of the cache, whether the cache should be prefilled with rows from the DW or be filled on-the-fly, whether full rows should be cached or only the keys and lookup attributes, and finally whether inserted rows should be put in the cache.A `CachedDimension` for the test dimension is shown in Listing 9.

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

Listing 9. Representing the test dimension using `CachedDimension`.

5.2 Advanced Dimension Support

`SlowlyChangingDimension` provides support for type 2 changes in a SCD [23] and in addition to type 2 changes, type 1 changes are also supported for a subset of the dimension's attributes. When an instance of `SlowlyChangingDimension` is created, it takes the same arguments 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 given. If given, the version of a row 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, 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 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 support for caching. Finally, it is possible to configure if rows should be sorted by the RDBMS such that `pygrametl` uses SQL's `ORDER BY` or if `pygrametl` instead should retrieve all versions of a row and sort them in Python. At least for one popular commercial RDBMS, it is significantly faster to do it in Python.

`SlowlyChangingDimension` offers the same methods as `Dimension`. `lookup` is, however, modified to return the key value for the *newest* version. To handle versioning, `SlowlyChangingDimension` offers the method `scdensure`. This method must be given a row (and optionally a name mapping). Like `ensure` it 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 occurred for other attributes, it creates a new version of the member and adds the new version to the dimension. Unlike the previously described methods, `scdensure` modifies the given row. Specifically, it adds the key and versioning values to the row so the user does not have to retrieve them afterward. The method `closecurrent` sets an end date for the current version without creating a new version.

In the running example, the web pages we test might be updated. To record this, we let the page dimension be a type 2 SCD. We lookup the page using the URL and detect changes by considering the other attributes. Using `SlowlyChangingDimension` to do so is shown in Listing 10.

In this example, 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 the user wants a different behavior, (e.g., that the *to date* is set to the day before the new member's from date), `tofinder` can be set to a function that performs the necessary calculations.

```
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')
```

Listing 10. Representing the page dimension using `SlowlyChangingDimension`.

`TypeOneSlowlyChangingDimension` is similar to `SlowlyChaningDimension` but *only* supports type 1 changes where dimension members are updated (not versioned) [23]. `SlowlyChaningDimension` also supports using type 1 changes, but they cannot be used alone, only together with type 2 changes. Also, by only supporting type 1 `TypeOneSlowlyChangingDimension` can be optimized more.

`SnowflakedDimension` supports filling a dimension in a snowflake schema [23]. 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. The member must thus be created by joining the tables in the snowflaked dimension together.

Normally, it can be a tedious task to create ETL logic for filling a snowflaked dimension. First, a lookup must 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. 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. While searching from the root is not difficult as such, it takes a lot of tedious coding which 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 user only has to make one method call to add/find the member.

An instance of `SnowflakedDimension` is constructed from other `Dimension` instances. The user creates a `Dimension` instance for each table participating in the snowflaked dimension and passes these instances to the `SnowflakedDimension` instance when creating. In the running example, the page dimension is snowflaked. We can create a `SnowflakedDimension` instance for the page dimension as shown in Listing 11 (where the different `Dimension` instances are created beforehand).

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

Listing 11. Representing the page dimension using `SnowflakedDimension`.

The argument is a list of pairs where the first element in each pair 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. 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. We could avoid both of these requirements but doing so would make **`SnowflakedDimension`** much more complicated to use. If the snowflake does not form a tree, the user 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 user 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 as shown in Listing 12.

```
# 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)
```

Listing 12. Ensuring that a row exists in a `SnowflakedDimension`

`SnowflakedDimension`'s **`lookup`** method calls the **`lookup`** method on the **`Dimension`** object at 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 downwards as much as needed. These two methods 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 how to track if an insertion did happen.

The algorithm is recursive and both **`ensure`** and **`insert`** first invoke it with *dimension* set to the root table. First a normal **`lookup`** is performed. If the key value is found, it is set in the row and returned (Lines 2–4). If not, the algorithm

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 $keyval$
```

is applied recursively on each of the tables that are referenced from the current table (Lines 5–6). As a side-effect 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` in Listing 11, we can use the SCD support on a snowflake. With a single call of `scdensure`, a full versioned 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 Pentaho Data Integration (PDI), the use of snowflakes requires the user to use several lookup/update steps. It is thus not easy to start looking up/inserting from the root as foreign key values might be missing. Instead, the user has to start from the leaves and go towards the root. In `pygrametl`, the user only has to call one method on a `SnowflakedDimension` instance. `pygrametl` starts at the root (to save lookups) and only if needed moves to the other levels.

`BulkDimension` is used in scenarios where a lot of data must be inserted into a dimension and it becomes too time-consuming to use traditional SQL `INSERT`s (like 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 RDBMS to RDBMS. Therefore, we require that the user passes a function that does the bulk loading when creating an instance of `BulkDimension`. This function is invoked by `pygrametl` when the bulkload should take place. When using the database driver psycopg2 [37] and the RDBMS PostgreSQL [36], the function can be defined as shown in Listing 13. Functions for more drivers are available in pygrametl's documentation [41].

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

Listing 13. A user-defined function for bulk-loading rows into PostgreSQL

The user can optionally define which separator and line-ending to use, which file to write the data to, and which string value to use for representing `NULL`.

To enable efficient lookups, `BulkDimension` caches all the dimension's data in memory. This is viable for most dimensions as modern computers have large amounts of memory. But if it is impossible and efficient bulk loading is desired `CachedBulkDimension` can be used. Like `CachedDimension`, the size of its cache can be configured, but it also supports bulk loading. To avoid code duplication, the code for bulk loading has been placed in the class `_BaseBulkloadable` which `BulkDimension` and `CachedBulkDimension` then both inherit from.

6 Fact Table Support

`pygrametl` provides four classes for representing fact tables. They all assume that a fact table has a number of key attributes that reference dimension tables. Further, the fact tables may also have a number of measure attributes.

`FactTable` provides a basic representation of a fact table. When an instance is created, the programmer must provide the fact table's name, the names of the key attributes, and optionally the names of the measure attributes if such exist.

`FactTable` provides the method `insert` which takes a row and inserts a fact into the DW's table. It also provides the 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 provides the method `ensure` which first tries to use `lookup`, and if a match is found, compares the measure values between the fact in the DW and the given row. It raises an error if these are different. If no match is found, it invokes `insert`. All the methods support name mappings.

`BatchFactTable` inherits from `FactTable` and provides the same methods. However, it caches inserted rows in memory until a user-configurable number of rows can be inserted together. This can significantly improve performance.

`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 written to a file, and when a user-configurable number of rows have been added to the file it is loaded into the fact table using code in `_BaseBulkloadable`. Like for `BulkDimension` and `CachedBulkDimension`, the user has to provide a function that performs the bulk-loading. For the running example, a `BulkFactTable` instance can be created as shown in Listing 14.

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

Listing 14. Representing the fact table (testresults) using `BulkFactTable`.

`AccumulatingSnapshotFactTable` provides a representation of a fact table where facts can be updated. In our example, a fact could be inserted when a test starts and then be updated with the result later. The users must specify the key attributes to use for lookup (`keyrefs`) and those that can be updated (`otherrefs`). The two sets must be disjoint. Updates are performed by `ensure` if a row with the same `keyrefs` exists, in which case the remaining attributes are set to the new row's values. A fact can also be explicitly updated using `update`.

7 Flow Support

A good aspect of graphical ETL tools is that it is easy to get an overview of an ETL flow. To make it easy to create small components with encapsulated functionality and connect such components, `pygrametl` provides support for *steps* and data flow. The user can, e.g., create a step for extracting data, a step for transforming data, and a step for loading the data into the DW's tables. The steps can be coded individually before the data flow between them is defined.

`Step` is the basic class for flow support. It can be used directly or as a base class for other step classes. For each `Step` the user can set a *worker function* which is applied to each row passing through the `Step`. If not set by the user, the method `defaultworker` (which does nothing) is used. Sub-classes of `Step` overrides `defaultworker`. The user 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 user specifies otherwise. So if no default `Step` is set or if the user wants to send the row to a non-default `Step` (e.g., for error handling), there is the method `_redirect` which the user can use to explicitly direct the row to a specific `Step`. `Step` also provides the method `_inject` which injects a new row into the flow before the current row is sent. The new row can be sent to a specific target, otherwise, the default will be used. This gives the user a great degree of flexibility.

`Step`s can modify the rows they are given. For example, `DimensionStep` calls `ensure` on a `Dimension` instance for each row it receives and adds the returned key to the row. Another example is `MappingStep` which applies functions to attributes values in 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, `ConditionalStep` is provided. A `ConditionalStep` is given a condition as a function. The function is applied to each row and if the result evaluates to `True`, the row is sent to the next default `Step`. In addition, another `Step` can optionally set as the target for rows where the condition evaluates to `False`. Otherwise, the row is silently discarded.

`Step`s can also perform aggregation. The base class for aggregating steps is `AggregatingStep`. Its `defaultworker` is called for each row it receives and must maintain the data needed to compute the final aggregate. This is done by the method `defaultfinalizer` which when given a row writes the aggregate to it.

All of this could be implemented without `Step`s. However, they are included for users who prefer connected components as provide by graphical ETL tools.

8 Testing Support

`pygrametl` provides *Drawn Table Testing (DTT)* to make testing of ETL flows easier. DTT simplifies defining tests where the state of the data sources and/or the DW (i.e., rows in tables of the database) must be taken into consideration. Currently, practitioners often store data in XML or CSV files separate from the tests and then add code to their tests that loads the data from these files into the test database, thus setting the test's *preconditions*. Likewise, the expected state is often also stored in text files and compared to the test database after the ETL flow has run, thus setting the test's *postconditions*. DTT makes it easy for users to define preconditions and postconditions as part of their tests as they can simply *draw* a table with rows using text shown in Listing 15. We call such a text-based representation of a database table a *Drawn Table (DT)*.

```
| testid:int (pk) | testname:text | testauthor:text |
| --------------- | ------------- | --------------- |
| -1              | Unknown       | Unknown         |
| 1               | Test1         | Søren           |
| 2               | Test2         | Christian       |
| 3               | Test3         | Torben          |
| 4               | Test4         | Ove             |
```

Listing 15. Example of a DT.

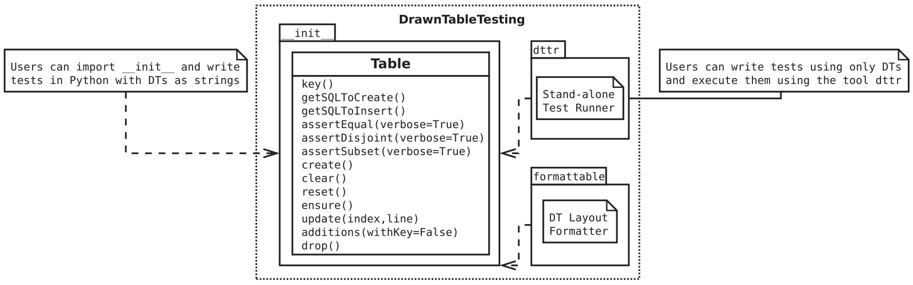

DrawnTableTesting
__init__
Table
key()
getSQLToCreate()
getSQLToInsert()
assertEqual(verbose=True)
assertDisjoint(verbose=True)
assertSubset(verbose=True)
create()
clear()
reset()
ensure()
update(index,line)
additions(withKey=False)
drop()
dttr
Stand-alone Test Runner
formattable
DT Layout Formatter
Users can import __init__ and write tests in Python with DTs as strings
Users can write tests using only DTs and execute them using the tool dttr

Fig. 3. Overview of the DTT package with public classes and intended use shown.

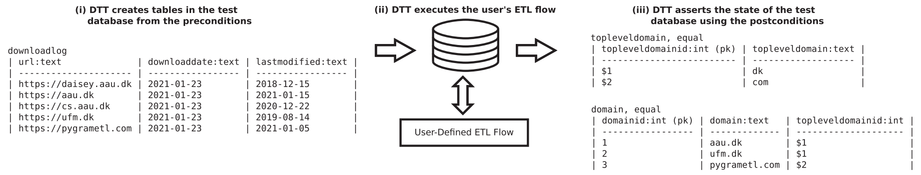
The DTT package consists of the DTT module and two command-line applications that use the module as shown in Fig. 3. By implementing the functionality of DTT as a module, interfaces for specific use-cases become easy to create.

Later, we will show how DTT can be used both from within Python (Sect. 8.5) and stand-alone using the test runner `dttr` (Sect. 8.6). `dttr` allows DTT to be used without the user has to write Python code. Also, while DTT is written in Python *it is not required* that users must implement their ETL flows in Python. Thus, the ETL flow can be implemented in another programming language or program (e.g., a graphical ETL tool).

8.1 Drawn Table (DT)

The DT in Listing 15 defines a table with three columns (of the shown types and of which `testid` is the primary key). The required syntax for a DT is as follows: the first row (called the *header*) contains `name:type` pairs for each column with each pair surrounded by vertical pipes. After a type, one or more constraints can be specified for the column as `(constraints)` with `(pk)` making the column (part of) the primary key. More information about constraints are provided in Sect. 8.2. If the table must hold any data, the header must be followed by a delimiter line containing only vertical pipes, spaces, and dashes (see Line 2) and then each row follows on a line of its own (see Line 3–7). Columns must be given in the same order as in the header and must be separated by pipes. For string values, any surrounding spaces are trimmed away. The syntax and types are checked by DTT's DT parser, while the constraints are checked by the RDBMS. A DT is also a valid table in GitHub Flavored Markdown [12].

How DTT tests an ETL flow using DTs is shown in Fig. 4. Before DTT can test the ETL flow, the user must create DTs that define the initial state of the test database (i.e., the preconditions) and/or DTs that define the expected state of the test database (i.e., the postconditions). Then DTT does the following: (i) Based on the DTs being set as preconditions, DTT automatically creates tables in a test database and loads the data from each DT into them. In this way, the initial state of the test database can easily be set without the user manually loading files or writing SQL. (ii) DTT then executes the user-defined ETL flow. (iii) Last, DTT verifies that the states of the postcondition DTs and the database tables they represent match. If they do not, DTT raises an informative error.


(i) DTT creates tables in the test database from the preconditions
(ii) DTT executes the user's ETL flow
(iii) DTT asserts the state of the test database using the postconditions
User-Defined ETL Flow

```
downloadlog
| url:text                 | downloaddate:text | lastmodified:text |
| ------------------------ | ----------------- | ----------------- |
| https://daisey.aau.dk    | 2021-01-23        | 2018-12-15        |
| https://aau.dk           | 2021-01-23        | 2021-01-15        |
| https://cs.aau.dk        | 2021-01-23        | 2020-12-22        |
| https://ufm.dk           | 2021-01-23        | 2019-08-14        |
| https://pygrametl.com    | 2021-01-23        | 2021-01-05        |
```

```
topleveldomain, equal
| topleveldomainid:int (pk) | topleveldomain:text |
| ------------------------- | ------------------- |
| $1                        | dk                  |
| $2                        | com                 |

domain, equal
| domainid:int (pk) | domain:text   | topleveldomainid:int |
| ----------------- | ------------- | -------------------- |
| 1                 | aau.dk        | $1                   |
| 2                 | ufm.dk        | $1                   |
| 3                 | pygrametl.com | $2                   |
```

Fig. 4. Using DTT to test a user-defined ETL flow.

As an example, assume the DT in Listing 15 is the postcondition in a test, but that the ETL flow is not defined yet. Thus, the test will fail since the test DW is empty. To make the test pass, we implement the part of the ETL flow that extracts, transforms, and loads the data into the table represented by the DT. As we make progress with the implementation, we can run the test again and again until the implementation passes the test. Thus DTT also enables test-driven development for ETL flows. From this small example, it is clear how DTT makes it simple to test an ETL flow during development, which helps to catch mistakes early such that they can be fixed immediately at low cost [13].

By default, DTT uses an in-memory SQLite database to run all tests against as it is very fast and does not require any installation or configuration by the user. Both of these properties are important as testing will be neglected if running the test suite is too time-consuming, thus allowing problems to accumulate and become harder to correct as additional changes to the ETL flow are performed [3, 29]. The user can, however, optionally use any other RDBMS with a PEP 249 connector (e.g., to run the test using the RDBMS used in production).

8.2 Constraints

As shown in Listing 15 a column can be made part of the primary key by specifying `(pk)` as a constraint. The constraints `unique` and `not null` are also supported and can be defined in the same manner. If multiple constraints are defined for one column, they must be separated by a comma. When representing multiple tables as DTs, foreign keys constraints are often required. In DTT, foreign keys must be specified as *fk target(att)* where *target* is the name of the referenced table and *att* is the referenced column. An example using foreign keys to connect `test` and `testresults` can be seen in Listing 16.

```
test
| testid:int (pk) | testname:text | testauthor:text |
| --------------- | ------------- | --------------- |
| -1              | Unknown       | Unknown         |
| 1               | Test1         | Søren           |
| 2               | Test2         | Christian       |
| 3               | Test3         | Torben          |
| 4               | Test4         | Ove             |

testresults
| testid:int (pk, fk test(testid)) | errors:int |
| -------------------------------- | ---------- |
| 1                                | 2          |
| 3                                | 0          |
```

Listing 16. Example of specifying foreign keys in a DT. Each table's name is above it. `pageid` and `dateid` are not shown to improve the examples readability.

8.3 Assertions

DTT can assert if a DT and a table in the test database are *equal*, *disjoint*, or if the DT is a *subset* of the table. As an example, assume that the user has specified that the DT in Listing 15 must be equal to the table in test database it represents. Thus, DTT verifies that the table in the test database contains the same rows as the DT (and only those) and if not, it raises an error and provides an easy-to-read explanation of why the test failed as shown in Listing 17.

```
AssertionError: test's rows differ from the rows in the database.
Drawn Table:
  | testid:int (pk) | testname:text | testauthor:text |
  | --------------- | ------------- | --------------- |
  | -1              | Unknown       | Unknown         |
  | 1               | Test1         | Søren           |
  | 2               | Test2         | Christian       |
  | 3               | Test3         | Torben          |
  | 4               | Test4         | Ove             |

Database Table:
  | testid:int (pk) | testname:text | testauthor:text |
  | --------------- | ------------- | --------------- |
  | 1               | Test1         | Søren           |
  | 2               | Test2         | Christian       |
  | 3               | Test3         | Torben          |
  | 4               | Test2         | Christian       |
  | -1              | Unknown       | Unknown         |

Violations:
  | testid:int (pk) | testname:text | testauthor:text |
  | --------------- | ------------- | --------------- |
E | 4               | Test4         | Ove             |
  |                 |               |                 |
D | 4               | Test2         | Christian       |
```

Listing 17. Assert equal for a table in the test database and the DT in Listing 15

In this example, the part of the user's ETL flow loading the table contains a bug. The DT instance in Listing 15 specifies that the dimension should contain a row for unknown tests and four rows for tests written by different people (see the expected state at the top of the output). However, the user's ETL code added `Test2` written by `Christian` a second time instead of `Test4` written by `Ove` (see the middle table in the output). To help the user quickly identify what rows do not match, DTT prints the rows violating the assertion which for equality is the difference between the two relations (bottom). The expected rows (i.e., those in the DT) are prefixed by an `E` and the rows in the database are prefixed by a `D`. Note that the orders of the rows can differ without making the test fail.

Asserting that a table in the test database and a DT are disjoint, makes it easy to verify that something *is not* in the database table, e.g., to test a filter or to check for the absence of erroneous rows that previously fixed bugs wrongly added. Asserting that a DT is a subset of a table in the test database, makes it easy to define a sample of rows that can be compared to a table containing so many rows that it is infeasible to embed them in a test. For example, it is easy to test if the leap day 2020–02–29 exists in the date dimension.

When compared to a table in the database, a DT's instance does not have to contain all of the database table's columns. Only the state of the columns included in the DT is then compared. This is useful for excluding columns for which the user does not know the state or for which the state does not matter in the test, like a generated primary key or audit information such as a timestamp.

8.4 Variables

Cases can also occur where attribute values must be equal, but the exact values are unknown or do not matter. A prominent case is when foreign keys are used.

In DTT this is easy to state using *variables*. A variable has a name prefixed by $ (the prefix is user-configurable) and can be used instead of attribute values in a DT. DTT checks if variables with the same name represent the same values in the database and fail the test if not. Listing 18 shows an example of how to use variables to test that foreign keys are assigned correctly.

```
tld
| tldid:int (pk) | tld:text           |
| -------------- | ------------------ |
| $1             | dk                 |
| $2             | org                |

domain
| domainid:int (pk) | domain:text   | tldid:int (fk tld(tldid)) |
| ----------------- | ------------- | ------------------------- |
| 1                 | aau.dk        | $1                        |
| 2                 | ufm.dk        | $1                        |
| 3                 | pygrametl.org | $2                        |
```

Listing 18. Check foreign keys using variables. Each table's name is above it. `topleveldomain` is shortened to `tld` to improve the examples readability.

Here the user has stated that the `tldid` of `aau.dk` in `domain` must match the `tldid` of `dk` in `tld`, likewise for `ufm.dk`, while `pygrametl.org` must match the `tldid` of `org`. If they do not, DTT raises errors as shown in Listing 19.

```
...
ValueError: Ambiguous values for $1; tld(row 0, column 0 tldid) is 1 and
    ↪  domain(row 0, column 2 tldid) is 2
...
```

Listing 19. Excerpt from the output of a test with mismatching variables.

The error message is from the output of a test case where `domain` and `tld` have the IDs defined in two different orders. As such, the foreign key constraints were satisfied although `aau.dk` is referencing the topleveldomain `org`. This shows that variables can test parts of an ETL flow that cannot be verified by foreign keys as foreign keys only ensure that a value is present.

It is also possible to specify that an attribute value should not be included in the comparison. This is done with the special variable `$_`. When compared to any value, `$_` is always considered to be equal. An example is shown in Listing 20 where the actual value of the primary key of the expected new row is not taken into consideration. `$_!` is a stricter version of `$_` which disallows `NULL`.

```
| topleveldomainid:int (pk)  | topleveldomain:text |
| -------------------------- | ------------------- |
| $_                         | dk                  |
| $_                         | org                 |
```

Listing 20. Example where a primary key value is ignored.

One limitation of variables is that a DT containing variables can only be used as a postcondition. The reason is of course that DTT does not know which values to insert into the database for the variables if the DT is used as a precondition.

8.5 Using Drawn Table Testing as a Python Package

DTT is easy to use as a Python package, i.e., as part of user-defined tests written in Python, e.g., using an existing testing framework like `unittest` or `pytest`.

The DT functionality is implemented by the `Table` class. To create an instance, a name for the table must be given as well as a string with a DT. An example of this is shown in Listing 21. Further, a string value representing `NULL` can optionally be given as well as an alternative prefix to be used for variables.

```
table = Table("test", """
| testid:int (pk) | testname:text | testauthor:text |
| --------------- | ------------- | --------------- |
| -1              | Unknown       | Unknown         |
| 1               | Test1         | Søren           |
| 2               | Test2         | Christian       |
| 3               | Test3         | Torben          |
| 4               | Test4         | Ove             |""")
table.ensure()
```

Listing 21. Example of creating and using a `Table` instance.

A DT's rows can also be loaded from an external source by providing either a path to a file containing a DT without a header or a data source to the constructor's `loadFrom` parameter. Data can thus be loaded from files, databases, etc., at the cost of the test not being self-contained. Users can of course represent multiple tables in the test database using multiple instances of `Table`.

After a `Table` instance is created, its `ensure` method can be executed. This will determine if a table with the same name and rows exists in the test database and otherwise create it (or raise an error if it contains other rows). If the user wants to create and fill the table even if it already exists, the `reset` method is provided. The `ensure` and `reset` methods raise an error if executed on a DT with variables as their values are unknown. The `create` method creates the table but does not load the rows. For users who prefer to create the table and insert the data represented by the DT themselves, the methods `getSQLToCreate` and `getSQLToInsert` are provided. DTT's asserts are implemented as the following methods: `assertEqual(verbose=True)`, `assertDisjoint(verbose=True)`, and `assertSubset(verbose=True)`. The detailed information they provide when a test fails (see Listing 17) can be disabled by executing them with the argument `False` for the parameter `verbose` (by default the argument is `True`).

`Table` instances are immutable. However, postconditions are often similar to preconditions except for a few added or updated rows. Therefore, we have made it simple to create a new `Table` instance from an existing one by appending or updating rows. Rows can be appended using the + operator, e.g., like `newTable = table + "| 5 | Test5 | Christian |" + "| 6 | Test6 | Ove |"`. The method `update(index, newrow)` creates a new instance with the values of the row at `index` changed to the values provided by `newrow`. For example, a copy of `table` with the first row changed, can be created as by `newTable = table.update(0, "| -1 | Unknown | N/A |")`. Note that a new instance of `Table` does not change the test database unless its `ensure`, `reset`, or `create` method is executed. By making `Table` instances immutable and creating new instances when they are modified, the user can very easily reuse the `Table` instance representing the precondition for multiple tests, and then as part of each test create a new instance with the postcondition based on it. After a number of additions and/or updates, it can be useful to get all modified rows. The user can then, e.g., make a test case where the ETL flow is executed for the new rows as shown in Listing 22.

```
1  def test_canInsertIntoTestDimensionTable(self):
2      expected = table + "| 5 | Test5 | Christian |" \
3                       + "| 6 | Test6 | Ove |"
4      newrows = expected.additions()
5      etl.executeETLFlow(newrows)
6      expected.assertEqual()
```

Listing 22. Example using the additions method.

In Listing 22, `expected` defines how the user expects the database state to become, but it is not the DTT package that puts the database in this state. The database is modified by the user's own ETL flow executed on Line 5. After the user's ETL flow have been executed, DTT verifies that the table in the test database and the DT are equal with `expected.assertEqual()` on Line 6.

A full example using DTT with Python's `unittest` module is shown in Listing 23. When using `unittest`, a class must be defined for each set of tests. We find it natural to group tests for a dimension into a class such that they can share DTs. A class using DTT to test the ETL flow for the test dimension is defined on Line 1. It inherits from `unittest.TestCase` as required by `unittest`. Two methods are then overridden: `setUpClass(cls)` and `setUp(self)`.

```
1  class TestStateTest(unittest.TestCase):
2      @classmethod
3      def setUpClass(cls):
4          cls.cw = dtt.connectionwrapper()
5          cls.initial = table = dtt.Table("test", """
6          | testid:int (pk) | testname:text | testauthor:text |
7          | --------------- | ------------- | --------------- |
8          | -1              | Unknown       | Unknown         |
9          | 1               | Test1         | Søren           |
10         | 2               | Test2         | Christian       |
11         | 3               | Test3         | Torben          |
12         | 4               | Test4         | Ove             |""")
13
14     def setUp(self):
15         self.initial.reset()
16
17     def test_insertNew(self):
18         expected = self.initial + "| 5 | Test5 | Christian |"
19         newrows = expected.additions()
20         etl.executeETLFlow(self.cw, newrows)
21         expected.assertEqual()
22
23     def test_insertExisting(self):
24         row = {testid:3, 'testname':'Test3', 'testauthor':'Torben'}
25         etl.executeETLFlow(self.cw, [row])
26         self.initial.assertEqual()
```

Listing 23. DTT tests for the test dimension.

The method `setUpClass(cls)` is executed before the tests are. The method requests a connection to a test database from DTT on Line 4 and defines a DT with the initial state of the dimension in Line 5. By creating them in `setUpClass(self)`, they are only initialized once and can be reused for each test. To ensure the tests do not affect each other, which would make the result depend on their execution order, the `test` table in the database is reset before each test by `setUp(self)`. On Line 17 and Line 23 the tests are implemented as methods prefixed by `test_`. `test_insertNew(self)` tests that a new row is inserted correctly, while `test_insertExisting(self)` tests that inserting an existing row does not create a duplicate. In this example, both of these tests execute the user's ETL flow by calling `executeETLFlow(connection, newrows)`.

8.6 Using Drawn Table Testing as a Stand-Alone Tool

The `dttr` (for DTT Runner) command-line application allows DTT to be used without doing any Python programming, i.e., with tests consisting only of DTs. Internally, `dttr` uses the DTT module and thus provides all of its functionality.

`dttr` uses *test files* (files with the suffix `.dtt`) to specify preconditions and/or postconditions. An example of a test file is given in Listing 24. Note that a test file does not contain any Python code. This file only contains one precondition (i.e., a DT with a name, but without an assert above it) on Line 1–4 and one postcondition (i.e., a DT with both a name and an assert above it) on Line 6–13). However, having both a precondition and a postcondition is not required, a `.dtt` file can contain any number of preconditions and/or postconditions. Users are free to structure their tests as they please. It is, e.g., possible to have a test file for the fact table and another test file for a dimension table and still ensure that an inserted fact's foreign key references a specific dimension member.

```
 1  test
 2  | testid:int (pk) | testname:text | testauthor:text |
 3  | --------------- | ------------- | --------------- |
 4  | -1              | Unknown       | Unknown         |
 5
 6  test, equal
 7  | testid:int (pk) | testname:text | testauthor:text |
 8  | --------------- | ------------- | --------------- |
 9  | -1              | Unknown       | Unknown         |
10  | 1               | Test1         | Søren           |
11  | 2               | Test2         | Christian       |
12  | 3               | Test3         | Torben          |
13  | 4               | Test4         | Ove             |
```

Listing 24. Example of a `dttr` test file.

The syntax for a precondition in a test file is as follows. On the first line of a precondition, the name of the table must be given, it is `test` in Listing 24. The default test database is used when only the name of the table is given. Alternatively, users can specify a PEP 249 connection in a configuration file and append `@` and the name of the connection to the table name, e.g., `test@targetdw`. After the table name, a DT must be specified (Lines 2–4). Preconditions and postconditions in a test file must be separated by an empty line (Line 5).

For a postcondition the name of the table must also be given first, followed by a comma and the assertion to use (Line 6). In the example, the table name is `test` like for the precondition, but they may be different. For example, the precondition could define the initial state for `inputdata@sourcedb` while the postcondition could define the expected state for `test@targetdw`. While the assertion `equal` is used in this example, DTT's other two assertions can also be used with: `disjoint` and `subset`. Then a DT is given like for the precondition (Lines 7–13). A DT used as part of a postcondition can contain variables.

For tests that require more data than what is feasible to embed in a DT, data from an external file or a database can be added to a DT by specifying a data source as its last line. For example, the line `csv DownloadLog.csv \t` adds the contents of DownloadLog.csv to the DT (using a tab the field separator) in addition to any rows drawn as part of the DT. By adding `sql oltp SELECT testid, testname, testauthor FROM test` as the last line, all rows of the table `test` from the PEP 249 connection `oltp` are added to the DT. This is user-extensible through a configuration file so users (or administrators) can add support for other sources of data, e.g., XML or a NoSQL DBMS like MongoDB.

`dttr` can be executed from the command-line as shown in Listing 25. Note that the ETL program to test and its arguments are given to `dttr` using `--etl`

```
dttr --etl "/Path/To/MyETLProgram --run MyFlow --loaddim test"
```

Listing 25. How to execute `dttr`.

When run, `dttr` by default looks for all `.dtt` files under the current working directory, but optional arguments allow the user to select which files to read. `dttr` then reads all these `.dtt` files. Then the preconditions from the files are set. This is done using `Table`'s `ensure` method. After the preconditions have been set, the user's ETL flow is executed. When the ETL flow has finished, all postconditions and variables are checked using `Table`'s `assert` methods.

9 Supporting Functionality

In addition to the modules and classes described in the previous sections (Sect. 4–8), `pygrametl` provides helpful supporting functionality that makes the modules and classes simpler to use and helps users parallelize their ETL flows.

9.1 Functions

`pygrametl` provides functions that operate on rows (e.g., to copy, rename, or project) and functions that convert types, but return a user-configurable default value if the conversion cannot be done (e.g., like `getfloat` shown in Listing 1).

Particularly 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 use 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 Listing 10. 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 the set of provided functions is relatively small, it is important to remember that the user has access to the many Python libraries that exist. Also, users can create private libraries with the functionality they often use.

9.2 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 [58]. The support was designed to preserve the simplicity of `pygrametl` programs. For example, extraction and transformation of data can run in another process using a `ProcessSource` as shown in Listing 26.

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

Listing 26. Transforming rows in a separate process using `ProcessSource`.

The `ProcessSource` spawns a process that extracts the `rawdata` and transforms it into `transformeddata` before assigning it to the `inputdata` variable in the main process. Similarly, `Dimension` instances can also run in another process, by means of `DecoupledDimension` which has the same interface as `Dimension` but pushes all the work to a `Dimension` instance in another process. This instance is given to `DecoupledDimension` when it is created as shown in Listing 27.

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

Listing 27. Update a dimension in another process using `DecoupledDimension`.

Work on a `FactTable` instance can also be performed in another process by using `DecoupledFactTable` as shown in Listing 28. Decoupled instances can 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 instance will get without involvement from the user (or main process). For details, see [58].

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

Listing 28. Update a fact table in another process using `DecoupledFactTable`.

Finally, `pygrametl` provides the `splitpoint` annotation which makes user-defined functions run in parallel with other tasks as shown in Listing 29.

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

Listing 29. Run a function in multiple processes using the `splitpoint`.

`pygrametl` uses processes instead of threads when running on CPython, the reference implementation of Python. The reason is that CPython cannot execute Python bytecode using multiple threads due to the global interpreter lock. However, `pygrametl` uses threads when running on Jython, an implementation of Python in Java, as the HotSpot JVM can execute multiple threads in parallel.

9.3 Editor Support

A key benefit of DTT is that users can easily understand the preconditions and postconditions of a test from the DTs. However, to gain the full benefit of DTs, their columns should be aligned across rows as they otherwise become very difficult to read. A very poorly formatted DT can be seen in Listing 30.

```
| testid:int (pk) | testname:text | testauthor:text |
| -------
    | -1              | Unknown             | Unknown         |
| 1                       | Test1         | Søren           |
                | 2 | Test2    | Christian          |
| 3                        | Test3         | Torben          |
| 4      | Test4         | Ove             |
```

Listing 30. Example of a very poorly formatted DT.

It is clear that poor formatting makes a DT harder to read. However, as properly formatting each DT can be tedious, DTT provides the `formattable` script which automates the task (see Fig. 3). It is designed to be used with extensible text editors so users can format a DT by placing the cursor on a DT and executing the script. Thus, integrating the script with the popular editors' Emacs and Vim requires only a few lines of Elisp and Vimscript, respectively.

10 Evaluation

To evaluate `pygrametl` and compare the development efforts for visual and code-based programming, an extended version of the original paper about `pygram-etl` [57] presented an evaluation where the running example was implemented in both `pygrametl` and the graphical ETL tool PDI [34], a popular 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. The implemented ETL flow and the data generator are available from pygrametl.org [43].

10.1 Productivity

It is obviously difficult to make a comparison of two such tools, and a full-scale test would require several teams of fully trained specialists, which is beyond our resources. We obviously know `pygrametl` well, but we also have significant experience with PDI from earlier projects. Each tool was used to create an identical solution twice. When creating the solution for the first time, we spent time on designing and implementing the ETL Flow. All the time we spent creating it again was *interaction time* (time spent on typing and clicking).

The `pygrametl`-based program was very easy to develop. It took a little less than an hour the first time, and 24 min the second time. The program consists of ∼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 in Listing 31.

```python
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()
```

Listing 31. Main method of the ETL flow made to measure development time.

The methods `extractdomaininfo` and `extractserverinfo` contain four lines of code to extract domain, top-level domain, and server name. Note that the page dimension is a SCD, where `scdensure` is an easy way to fill both a snowflaked and a SCD. The date dimension uses a `rowexpander` passed to the `datedim` object to (on demand) calculate the attribute values so `ensure` can find or insert a member. The test dimension is preloaded, and we only do lookups.

The PDI-based solution took us a little more than two hours to create the first time, and 28 min the second time. The flow is shown in Fig. 5.

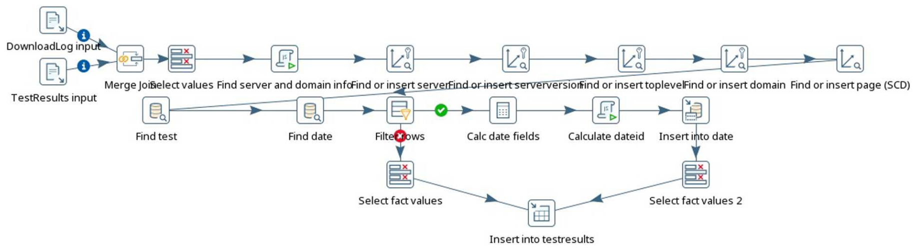


Fig. 5. Data flow in PDI-based solution.

We emulate the `rowexpander` feature of `pygrametl` by first looking up a date and then 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 it is far simpler to work out in `pygrametl` (compare the main method and Fig. 5). In addition, it was a little faster to create the solution a second time by typing code compared to click around in the GUI. So, while experienced PDI users may be able to create a solution as fast in PDI, we believe that `pygrametl` is simpler and easier to use.

10.2 Performance

The extended version of the original paper [57] also presented performance results for both PDI and `pygrametl` on the running example. In this paper, we provide new results for the same running example, but with newer versions of both PDI (version 7.1) and `pygrametl` (version 2.5) when 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 ∼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 virtual machine (we did not test PDI's support for distributed execution) with three virtual processors, and 16 GB of RAM (the CPUs were never completely used during the experiments and the amount of RAM was large enough to allow all of the dimension data to be cached). 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 32 GB of RAM, SSD disk, and a 2.70 GHz Intel i7 CPU with 4 cores and hyperthreading.

We used 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 bulk loading 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 bulk loading of facts (using `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 maximum Java heap size was set to 12 GB.

Figure 6(a) shows the elapsed wall-clock time for the loads and Fig. 6(b) shows the spent CPU time. It can be seen that the amounts of time grow linearly for both PDI and `pygrametl`. It is however clear that `pygrametl` is significantly

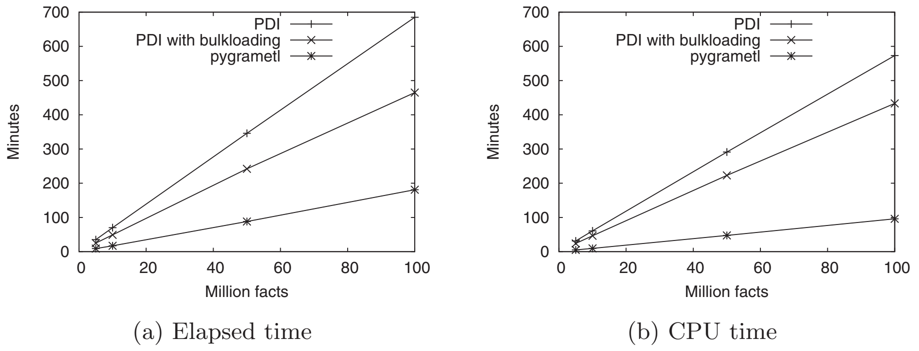


Fig. 6. Performance results.

faster than PDI in this experiment. When loading 100 million facts, the `pygrametl`-based solution handles 9208 facts/s. PDI with a single connection handles 2433 and PDI with two connections handles 3584 facts/s. Also, while servers often have many CPUs/cores it is still desirable if the ETL uses little CPU time, so more CPU time is available for other purposes like processing queries. This is particularly relevant if the ETL is running on a virtualized server with contention for the CPUs. From Fig. 6(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 clear to see that in terms of resource consumption; it is beneficial to create a light-weight specialized program instead of using a general feature-rich but heavy-weight ETL application.

11 Case Study

The company FlexDanmark 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 are 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 using `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 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 be tracked in a similar way to how dimension changes are tracked for a type-2 SCD. Further, new sources and dimensions are sometimes added. The ETL procedure for the DW is also implemented in Python, but FlexDanmark has created a framework that using *templates can generate Python code* incl. `pygrametl` objects based on metadata parameters [2]. 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 flows for these things in traditional graphical tools was found to be hard. FlexDanmark did 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 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 graphical ETL tool, but after comparing a few programmatic solutions with their counterparts created using the graphical tool, it was found to be faster and more flexible to code the ETL flows.

In most cases where `pygrametl` is used, we are not aware for what since users are not obliged to register in any way. Some `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 so far; we have been unable to extract guidelines or principles for how best to design programmatic ETL flows. 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 be those already used in the organization. Reusing existing development principles should also reduce the time required to learn `pygrametl` as users then only have to learn a framework and not new development principles.

12 Experiences from Open-Sourcing pygrametl

When we published the first paper about `pygrametl`, we also made the source code available on our department's web page. From the logs, we could see that there were some downloads and we also received a few questions and comments.

Later, the source code was moved to Google Code, and the attention it received increased. When Google Code was closed, the code was moved to GitHub [39] where it got a lot more attention. The lesson we learned from this is that it is very important to publish source code at the places where people are used to looking for open source projects. Before we moved to GitHub, others had already created unofficial and unmaintained repositories with the `pygrametl` code on GitHub, just so it was available there. Similarly, we learned that it should be easy for users to install the framework. Thus, it needs to be available from the Python Package Index (PyPI) [40]. Again we experienced that unofficial and unmaintained copies were created before we published an official version. With `pygrametl` on PyPI it can easily be installed using the pip package manager by executing the following command on the command-line: `pip install pygrametl`.

As the number of changes we received from other users increased, it became harder to ensure that the existing code did not break without extending our testing procedures. To simplify this, we developed DTT. During development, it became clear it could be used to not just test `pygrametl` but also ETL flows.

Another lesson we learned, although not very surprising, is that good documentation is needed. By making a Beginner's Guide and examples available online [42], we have reduced the number of (repeated) questions to answer via e-mail and made it much easier to get started with `pygrametl`. It is also important to clearly describe what the tool is intended to do. For example, we have received multiple questions about how to do general data movement from one platform to another, despite `pygrametl` being an ETL framework for *dimensional* DWs.

We have also received some excellent contributions from users. They have, e.g., found minor mistakes in corner cases of the framework, removed performance bottlenecks, helped generalize features, and in general improved the functionality of `pygrametl`. An interesting example is `useorderby`. When performing a lookup `SlowlyChangingDimension` used to only retrieve the newest version of a row by first sorting them using `ORDER BY`. However, for some RDBMSs, a user found that transferring all versions of a row and selecting the newest version in the Python process provided significantly better performance. So, the parameter `useorderby` was added to `SlowlyChangingDimension` so users can choose which behavior they want. Similarly, a user found that `BatchFactTable` performed poorly with some RDBMSs due to its use of `executemany`. We found that some PEP 249 connectors implement `executemany` as a loop around `execute` or raises an error. So we added support for loading batches using one SQL statement run by `execute` instead of multiple SQL statements run by `executemany`.

Over time it has also become clear that some of the functionality provided by `pygrametl` is not being used. For examples, `Step`s seems to not be a useful feature for our users. While we have received comments, questions, and bug reports about most of the other functionality provided by `pygrametl`, we have received virtually nothing about `Step`s. It seems that users who choose to use programmatic ETL prefer not to think in the terms required by the graphical ETL tools. Thus, ideas from `Step`s have been reused in other parts of `pygrametl`, e.g., `MappingStep` was re-implemented as the `MappingSource` data source.

`pygrametl` is published under a 2-Clause BSD license. We chose this license as it is very permissive. The license allows proprietary use of the 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 contributions from users despite them not being forced to by the license.

13 Related Work

13.1 Data Integration Tools

Many commercial ETL and data integration tools exist [4] and many methods for creating ETL flows have been proposed [1]. Among the vendors of the most popular products, we find big players like IBM, Informatica, Microsoft, Oracle, and SAP [20,21,30,32,46]. These vendors and many others provide powerful tools supporting integration between different kinds of sources and targets based on graphical design of the flows. Due to their wide field of functionality, the commercial tools often have steep learning curves and as previously mentioned, 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 [55]. In some open source ETL tools, such as the popular PDI [34] and Talend [49], the user also specifies the ETL flow in a GUI. Scriptella [47] is an example of a tool where the ETL flow 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 SCDs and snowflake schemas, to name a few.

The academic community has also been attracted to ETL, with multiple surveys published about the research [1,61] Most of the academic approaches, e.g., [48,60], use graphs or UML to model an ETL workflow. We challenge the idea that graphical programming of ETL is always easier than text-based programming. Grönniger et al. [16] 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 [35] 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 [35], she found that text overall was faster to use than graphics. Andersen et al. [2] also argue for using programmatic ETL and propose the framework SimpleETL which reduces the amount of boilerplate code required for `pygrametl` programs by generating code from a metadata specification.

13.2 Parallelism

As ETL tools often operate on very large amounts of data, parallelization is widely applied in ETL tools. Parallel execution can sometimes be achieved without the user doing anything to enable it. In PDI [34], for example, each *step* (shown as a box in the graphical representation of the ETL flow) executes in its own thread. It is also possible to define that several instances of a step should run in parallel and have the rows distributed among them in round robin fashion. In Microsoft's popular ETL tool SSIS [30] many threads are also used to execute a data flow. SSIS uses a more advanced policy than PDI and considers *execution trees* of transformations that can be processed by the same thread. An execution tree can be assigned one or more threads depending on its CPU utilization. Further, parameters can be used to limit the number of parallel threads. However, it is very different how well different designs for an ETL flow exploits parallelism. Therefore, how a tool applies parallelism must also be considered carefully when a graphical ETL tool is used.

ETLMR [25,26] modifies `pygrametl` to be used with MapReduce [6] on a cluster. With ETLMR it is thus easy to create scalable dimensional ETL solutions on MapReduce with few lines of code. CloudETL [27] is a Java-based framework for creating dimensional ETL flows that load data into DWs managed by Hive [59] using MapReduce. The user specifies the ETL flow using high-level constructs while CloudETL takes care of the parallelization. CloudETL offers support for efficient individual lookups, inserts, and updates of SCDs and thus shares some ideas with ETLMR and `pygrametl`.

In contrast to the graphical ETL tools [30,34], the user has complete control over the parallelization in `pygrametl` and can decide which parts should execute in parallel. The support for `TransformingSource` and `ProcessSource` makes it possible to group certain transformations together and run them in a separate thread or process. Also, the support for `Decoupled` objects (`DecoupledDimension` and `DecoupledFactTable`) makes it easy to process (some or all) dimensions and fact tables in parallel. The parallelization constructs provided by `pygrametl` allow the ETL flow to run on a single server with multiple cores. This does not scale as much as the cluster-based solutions [25–27] but does also have less overhead.

13.3 Testing

While many methods have been proposed for creating ETL flows and DWs in general, research about ensuring their correctness is much more limited [5]. As existing literature already has been surveyed by ElGamal et al. [10], Chandra and Gupta [5], and Homayouni et al. [19], we only summarize selected publications on testing strategies and tools for testing ETL flows.

Golfarelli and Rizzi [13] provide a comprehensive discussion of all aspects of DW testing with ETL testing described as the most complex and critical testing phase. They propose using not only real correct data for ETL test but also data simulating errors. In their following work, they propose how their approach can be used with an iterative DW design process, and they further validate their approach through a case study [14,15]. ElGamal et al. [11] propose an architecture-agnostic testing strategy that, as a result, can be used with existing DWs. A testing manual accompanies the paper with a detailed description of each test to guide users of the strategy [9]. However, the paper states that an evaluation discovered the need to point to tools that could be used to automate the tests. ETLDiff [54] is an approach for regression testing of an entire ETL flow. The tool explores the DW schema and executes a query that joins a fact table to its dimension tables. The first time this is done, the result is stored, and the results of subsequent executions are then compared to this expected result and any differences are pointed out. In the comparison, the actual values of surrogate keys are ignored while the tool still considers if facts are referencing the right dimension members. Guduric [17] describes that it is common for SSIS users to do testing by running their ETL flow on a sample database and then manually sample the outcome. To make this more effective and reliable, he proposes the tool SSISTester that allows SSIS packages to be tested using an interface similar to that used by the Visual Studio testing framework. The tool requires tests to be written in C#. Like with traditional unit testing, it is the responsibility of the tester to write code to set up the precondition, compare the achieved state with the desired outcome, and clean up. Marin [28] proposes a method for testing OLAP cubes using the source OLTP database as the truth. The approach is implemented in Microsoft Dynamics AX. To test a cube, MDX queries are executed on it with equivalent T-SQL queries executed on the OLTP source database. The MDX queries are created based on a coverage model so redundant queries are not included in the test set and the T-SQL queries are automatically generated from the MDX queries. While this approach is data-agnostic, due to the limitations of their query translation method, it is mostly suitable for cases where little to no transformation is performed on the source data when loaded. A tool developed at Cambia Health Solutions [22] executes SQL queries against a test database and validates the output against an expected result. It was specifically designed for staff with a strong SQL background but who lacked knowledge of general-purpose programming languages like Java and Python. Bijoux [31,51] is a tool for generating test data for ETL flows. The input to Bijoux is an ETL flow and the output is data sets generated by considering the different paths and constraints in the ETL flow. Homayouni et al. [18] propose testing the entire ETL flow by comparing properties (e.g., number of rows) of the data in the sources and the DW when loaded. Tests can automatically be generated to check these properties if mappings from the sources and the DW are available in a structured manner. For example, the paper demonstrates how the mappings can be extracted from definitions of SQL views.

Compared to DTT, multiple papers focus on methodology and do not present any concrete tools despite commonly stating the benefit of automation and testing early in the DW design process [9,11,13–15]. The proposed ETL testing tools either only test the finished DW and not the ETL flow during development [18,22,28,54], require that the test environment is manually prepared [17,22,54], require that the user knows how to program [17], or only support asserting equality [22,28,54] Last, the focus of Bijoux [31,51] is on evaluating quality characteristics (performance, completeness, etc.) and not on improving productivity and correctness like DTT. From this it is clear that DTT fulfills a niche not covered by the other tools by making it simple to test an ETL flow by *drawing* preconditions and postconditions as easy-to-read DTs.

14 Conclusion and Future Work

We have presented an up-to-date overview of ***pygrametl***, a framework for programmatic ETL. We challenge the conviction that ETL flows should always be developed using a graphical ETL tool. We instead propose to let *ETL developers* (which are typically specialists) create ETL flows *by writing code*. To make this easy, `pygrametl` provides commonly used functionality, such as reading from data sources and filling 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. Flow support allows users to implement their ETL flows using connected steps. `pygrametl` also simplifies testing ETL flows through *Drawn Table Testing* which makes it simple for users to specify preconditions and postconditions for an ETL flow using string-based *Drawn Tables*. Last, `pygrametl` provides functions for common transformations and simple abstractions that make it easy for users to parallelize their ETL flows.

Our evaluation and case study show that creating ETL flows with `pygrametl` provides high programmer productivity and that the created ETL flows have good run-time performance. `pygrametl`'s flexible support for fact and dimension tables makes it easy to fill the DW. Thus the user can concentrate on the transformations needed for the data where they benefit from 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. By making the framework open source, we have learned the importance of publishing code at well-known places such as GitHub and the joy of users contributing improvements in the form of code, documentation, and ideas for how to improve `pygrametl`.

As future work, we are considering introducing 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. Also, the way to load rows (plain SQL `INSERT`s, batched `INSERT`s, or by bulk loading) is currently defined by the individual classes. A more general approach could be by composition of loader classes into the classes that handle dimensions and fact tables. It would also be interesting to investigate if specialized code can be generated using a set of templates from which the user can select the features they require, such as bulk loading. This could potentially provide a big performance advantage if, e.g., branches could be eliminated. We are still gaining experiences with DTT, but already see several directions for future work. For the variables, it would be useful if expressions could be used such that users, e.g., could specify an expected value as `$1 + 1` or define a variable using the state of another `$2 = $1 / 86400`. Another interesting direction is to determine if other kinds of asserts could be added to enable easy specification of a wider range of tests. It would also be relevant to add better support for other kinds of tests to the framework (e.g., stress testing and performance testing) to create a single complete test automation tool for DW projects. It would also be interesting to determine if specifying and modifying DTs can be simplified through better editor support and database integration. Last, a strength of `pygrametl` is the easy integration with other Python projects. So better integration with relevant projects such as Pandas [33] and/or PyArrow [38] would also be beneficial.

Acknowledgments. This research was, in part, supported by the DiCyPS center funded by Innovation Fund Denmark [7], the GOFLEX project funded by European Union grant number 731232 [50], and the SEMIOTIC project funded by Danish Council for Independent Research grant number 8022–00284B. We also thank all who helped improve `pygrametl`, particularly code and documentation contributors as well as users who provided feedback.

References

1. Ali, S.M.F., Wrembel, R.: From conceptual design to performance optimization of ETL workflows: current state of research and open problems. VLDB J. (VLDBJ) **26**(6), 777–801 (2017). https://doi.org/10.1007/s00778-017-0477-2
2. Andersen, O., Thomsen, C., Torp, K.: SimpleETL: ETL processing by simple specifications. In: 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP). CEUR-WS.org (2018)
3. Beck, K.: Test Driven Development: By Example, pp. 194–195. Addison-Wesley Professional, Boston (2002)
4. Beyer, M.A., Thoo, E., Selvage, M.Y., Zaidi, E.: Gartner magic quadrant for data integration tools (2020)
5. Chandra, P., Gupta, M.K.: Comprehensive survey on data warehousing research. Int. J. Inf. Technol. (IJIT) **10**(2), 217–224 (2018). https://doi.org/10.1007/s41870-017-0067-y
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: 6th Operating Systems Design and Implementation (OSDI), pp. 137–150. USENIX (2004)
7. DiCyPS - Center for Data-Intensive Cyber-Physical Systems. https://www.dicyps.dk/dicyps-in-english/. Accessed 10 Feb 2021
8. Django. https://djangoproject.com/. Accessed 10 Feb 2021
9. ElGamal, N.: Data warehouse test routine descriptions. Technical report, Cairo University (2016). https://doi.org/10.13140/RG.2.1.3755.5282
10. ElGamal, N., El Bastawissy, A., Galal-Edeen, G.: Towards a data warehouse testing framework. In: 2011 Ninth International Conference on ICT and Knowledge Engineering (ICT&KE), pp. 65–71. IEEE (2012). https://doi.org/10.1109/ICTKE.2012.6152416

11. ElGamal, N., El-Bastawissy, A., Galal-Edeen, G.H.: An architecture-oriented data warehouse testing approach. In: 21st International Conference on Management of Data (COMAD), pp. 24–34. CSI (2016)
12. GitHub Flavored Markdown Spec - Version 0.29-gfm, 06 April 2019. https://github.github.com/gfm/#tables-extension-. Accessed 10 Feb 2021
13. Golfarelli, M., Rizzi, S.: A comprehensive approach to data warehouse testing. In: 12th International Workshop on Data Warehousing and OLAP (DOLAP), pp. 17–24. ACM (2009). https://doi.org/10.1145/1651291.1651295
14. Golfarelli, M., Rizzi, S.: Data warehouse testing. Int. J. Data Warehous. Min. (IJDWM) **7**(2), 26–43 (2011). https://doi.org/10.4018/jdwm.2011040102
15. Golfarelli, M., Rizzi, S.: Data warehouse testing: a prototype-based methodology. Inf. Softw. Technol. (IST) **53**(11), 1183–1198 (2011). https://doi.org/10.1016/j.infsof.2011.04.002
16. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.S.: Text-based modeling. In: 4th International Workshop on Language Engineering (ATEM) (2007)
17. Guduric, P.: SQL server - unit and integration testing of SSIS packages. MSDN Mag.: Microsoft J. Dev. **28**(8), 48–56 (2013). http://download.microsoft.com/download/a/3/1/a315bac2-8093-45fd-8d04-1a9f899aca53/mdn_0813dg.pdf
18. Homayouni, H., Ghosh, S., Ray, I.: An approach for testing the extract-transform-load process in data warehouse systems. In: 22nd International Database Engineering & Applications Symposium (IDEAS), pp. 236–245. ACM (2018). https://doi.org/10.1145/3216122.3216149
19. Homayouni, H., Ghosh, S., Ray, I.: Data warehouse testing. Adv. Comput. **112**, 223–273 (2019). https://doi.org/10.1016/bs.adcom.2017.12.005
20. IBM InfoSphere DataStage. https://www.ibm.com/ms-en/marketplace/datastage. Accessed 10 Feb 2021
21. Informatica. https://informatica.com. Accessed 10 Feb 2021
22. Iyer, S.: Enabling ETL test automation in solution delivery teams. In: Excerpt from PNSQC 2014 Proceedings, pp. 1–10. PNSQC.org (2014)
23. Kimball, R., Ross, M.: The Data Warehouse Toolkit, 2nd edn. Wiley, New York (2002)
24. Kimball, R., Ross, M., Thornthwaite, W., Mundy, J., Becker, B.: The Data Warehouse Lifecycle Toolkit, 2nd edn. Wiley, Indianapolis (2008)
25. Liu, X., Thomsen, C., Pedersen, T.B.: ETLMR: a highly scalable dimensional ETL framework based on MapReduce. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2011. LNCS, vol. 6862, pp. 96–111. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23544-3_8
26. Liu, X., Thomsen, C., Pedersen, T.B.: ETLMR: a highly scalable dimensional ETL framework based on MapReduce. In: Hameurlain, A., Küng, J., Wagner, R., Cuzzocrea, A., Dayal, U. (eds.) Transactions on Large-Scale Data- and Knowledge-Centered Systems VIII. LNCS, vol. 7790, pp. 1–31. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37574-3_1
27. Liu, X., Thomsen, C., Pedersen, T.B.: CloudETL: scalable dimensional ETL for hive. In: 18th International Database Engineering & Applications Symposium (IDEAS), pp. 195–206. ACM (2014). https://doi.org/10.1145/2628194.2628249
28. Marin, M.: A data-agnostic approach to automatic testing of multi-dimensional databases. In: 7th International Conference on Software Testing, Verification and Validation (ICST), pp. 133–142. IEEE (2014). https://doi.org/10.1109/ICST.2014.26
29. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship, pp. 132–133. Pearson Education, London (2009)

30. Microsoft SQL Server Integration Services. https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services. Accessed 10 Feb 2021
31. Nakuçi, E., Theodorou, V., Jovanovic, P., Abelló, A.: Bijoux: data generator for evaluating ETL process quality. In: 17th International Workshop on Data Warehousing and OLAP (DOLAP), pp. 23–32. ACM (2014). https://doi.org/10.1145/2666158.2666183
32. Oracle Data Integrator. https://www.oracle.com/technetwork/middleware/data-integrator/overview/index.html. Accessed 20 Feb 2021
33. Pandas. https://pandas.pydata.org/. Accessed 10 Feb 2021
34. Pentaho Data Integration - Kettle. https://github.com/pentaho/pentaho-kettle. Accessed 10 Feb 2021
35. Petre, M.: Why looking isn't always seeing: readership skills and graphical programming. Commun. ACM (CACM) **38**(6), 33–44 (1995). https://doi.org/10.1145/203241.203251
36. PostgreSQL. https://postgresql.org. Accessed 10 Feb 2021
37. psycopg. https://www.psycopg.org/. Accessed 10 Feb 2021
38. PyArrow. https://pypi.org/project/pyarrow/. Accessed 10 Feb 2021
39. pygrametl - GitHub. https://github.com/chrthomsen/pygrametl. Accessed 10 Feb 2021
40. pygrametl - PyPI. https://pypi.org/project/pygrametl/. Accessed 10 Feb 2021
41. pygrametl.org - Bulk Loading. http://pygrametl.org/doc/examples/bulkloading.html. Accessed 10 Feb 2021
42. pygrametl.org - Documentation. http://pygrametl.org/doc/index.html. Accessed 10 Feb 2021
43. pygrametl.org - ETL Flow and Data generator. http://pygrametl.org/assets/pygrametlexa.zip. Accessed 10 Feb 2021
44. Python. https://python.org. Accessed 10 Feb 2021
45. Ruby on Rails. https://rubyonrails.org/. Accessed 10 Feb 2021
46. SAP Data Services. https://www.sap.com/products/data-services.html. Accessed 10 Feb 2021
47. Scriptella. https://scriptella.org. Accessed 10 Feb 2021
48. Simitsis, A., Vassiliadis, P., Terrovitis, M., Skiadopoulos, S.: Graph-based modeling of ETL activities with multi-level transformations and updates. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 43–52. Springer, Heidelberg (2005). https://doi.org/10.1007/11546849_5
49. Talend Open Studio for Data Integration. https://www.talend.com/products/data-integration/data-integration-open-studio/. Accessed 10 Feb 2021
50. The GoFlex Project. https://goflex-project.eu/. Accessed 10 Feb 2021
51. Theodorou, V., Jovanovic, P., Abelló, A., Nakuçi, E.: Data generator for evaluating ETL process quality. Inf. Syst. (IS) **63**, 80–100 (2017). https://doi.org/10.1016/j.is.2016.04.005
52. Thomsen, C., Andersen, O., Jensen, S.K., Pedersen, T.B.: Programmatic ETL. In: Zimányi, E. (ed.) eBISS 2017. LNBIP, vol. 324, pp. 21–50. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96655-7_2
53. Thomsen, C., Pedersen, T.B.: Building a web warehouse for accessibility data. In: 9th International Workshop on Data Warehousing and OLAP (DOLAP), pp. 43–50. ACM (2006). https://doi.org/10.1145/1183512.1183522
54. Thomsen, C., Pedersen, T.B.: ETLDiff: a semi-automatic framework for regression test of ETL software. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2006. LNCS, vol. 4081, pp. 1–12. Springer, Heidelberg (2006). https://doi.org/10.1007/11823728_1

55. Thomsen, C., Pedersen, T.B.: A survey of open source tools for business intelligence. Int. J. Data Warehous. Min. (IJDWM) **5**(3), 56–75 (2009). https://doi.org/10.4018/jdwm.2009070103
56. Thomsen, C., Pedersen, T.B.: pygrametl: a powerful programming framework for extract-transform-load programmers. In: 12th International Workshop on Data Warehousing and OLAP (DOLAP), pp. 49–56. ACM (2009). https://doi.org/10.1145/1651291.1651301
57. Thomsen, C., Pedersen, T.B.: pygrametl: a powerful programming framework for extract-transform-load programmers. Technical report, Aalborg University (2009). http://dbtr.cs.aau.dk/DBPublications/DBTR-25.pdf
58. Thomsen, C., Pedersen, T.B.: Easy and effective parallel programmable ETL. In: 14th International Workshop on Data Warehousing and OLAP (DOLAP), pp. 37–44. ACM (2011). https://doi.org/10.1145/2064676.2064684
59. Thusoo, A., et al.: Hive - a petabyte scale data warehouse using hadoop. In: 26th International Conference on Data Engineering (ICDE), pp. 996–1005. IEEE (2010). https://doi.org/10.1109/ICDE.2010.5447738
60. Trujillo, J., Luján-Mora, S.: A UML based approach for modeling ETL processes in data warehouses. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 307–320. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39648-2_25
61. Vassiliadis, P.: A survey of extract-transform-load technology. Int. J. Data Warehous. Min. (IJDWM) **5**(3), 1–27 (2009). https://doi.org/10.4018/jdwm.2009070101