pygrametl Documentation

Release 2.7

Aalborg Universitet

Dec 05, 2022

CONTENTS

pygrametl is a package for creating Extract-Transform-Load (ETL) programs in Python.

The package contains several classes for filling fact tables and dimensions (including snowflaked and slowly changing dimensions), classes for extracting data from different sources, classes for optionally defining an ETL flow using steps, classes for parallelizing an ETL flow, classes for testing an ETL flow, and convenient functions for often-needed ETL functionality.

The package's modules are:

- **datasources** for access to different data sources
- **tables** for giving easy and abstracted access to dimension and fact tables
- **parallel** for parallelizing an ETL flow
- **JDBCConnectionWrapper** and **jythonmultiprocessing** for Jython support
- **aggregators** for aggregating data
- **steps** for defining steps in an ETL flow
- **FIFODict** for a dict with a limited size and where elements are removed in first-in, first-out order
- **drawntabletesting** for testing an ETL flow

pygrametl is currently being maintained at Aalborg University in Denmark by the following people:

Current Maintainers

- Christian Thomsen <chr@cs.aau.dk>
- Søren Kejser Jensen <devel@kejserjensen.dk>

Former Maintainers

- Christoffer Moesgaard <cmoesgaard@gmail.com>
- Ove Andersen <ove.andersen.oa@gmail.com>

CHAPTER

ONE

GETTING STARTED

1.1 Install Guide

Installing pygrametl is fairly simple, mainly due to the package having no mandatory dependencies. This guide contains all the information needed to install and use the package with CPython. pygrametl also supports the JVM-based Python implementation Jython. For more information about using pygrametl with Jython see *Jython*.

1.1.1 Installing a Python Implementation

pygrametl requires an implementation of the Python programming language to run. Currently, pygrametl officially supports the following implementations (other implementations like PyPy and IronPython might also work):

- Jython, version 2.7 or above
- Python 2, version 2.7 or above
- Python 3, version 3.4 or above

Warning: As Python 2 is no longer being maintained support for it will slowly be reduced as we continue to develop pygrametl. Currently, `dttr` is the only pygrametl module that requires Python 3 (version 3.4 or above).

After a Python implementation has been installed and added to the system's path, it can be run from either the command prompt in Windows or the shell in Unix-like systems. This should launch the Python interpreter in interactive mode, allowing commands to be executed directly on the command line.

```
Python 3.9.2 (default, Feb 20 2021, 18:40:11)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

1.1.2 Installing pygrametl

pygrametl can either be installed from PyPI using a package manager, such as pip or conda, or by manually checking out the latest development version from the official GitHub repository. Installing pygrametl from PyPI is currently the simplest way to install pygrametl as the process is automated by the package manager. Bug fixes and new experimental features are, however, of course, available first in the GitHub repository.

Install from PyPI with pip

pip can install pygrametl to the Python implementation's global package directory, or to the user's local package directory which is usually located in the user's home directory. Installing pygrametl globally will often require root or administrator privileges with the advantage that the package will be available to all users of that system. Installing it locally will only make it available to the current user, but the installation can be performed without additional privileges. The two types of installation can be performed using one of the following commands:

```
# Install pygrametl to the global package directory
$ pip install pygrametl

# Install pygrametl to the user's local package directory
$ pip install pygrametl --user
```

Install from PyPI with conda

conda is an alternative package manager for Python. It is bundled with the Anaconda CPython distribution from Anaconda, Inc. There is no official pygrametl conda package as it uses a different package format than pip. It is however trivial to download, convert, and install the PyPI package using conda with only a few commands.

```
# Create a template for the conda package using the PyPI package
$ conda skeleton pypi pygrametl

# Build the conda package
$ conda build pygrametl/meta.yaml

# Install the conda package
$ conda install --use-local pygrametl
```

Afterward, the folder containing the package template can be deleted as it is only used for building the package.

Install from GitHub

The latest development version of pygrametl can be downloaded from the official GitHub repository. The project currently uses Git for version control, so the repository can be cloned using the following command.

```
# Clone the pygrametl repository from GitHub
$ git clone https://github.com/chrthomsen/pygrametl.git
```

Before Python can import the modules, the pygrametl package must be added to `sys.path`. This can be done manually in your Python programs, by setting `PYTHONPATH` if CPython is used, or by setting `JYTHONPATH` if Jython is used. More information about how CPython and Jython locate modules can be found in the two links provided.

1.1.3 Verifying Installation

A simple way to verify that pygrametl has been installed correctly and is accessible to the Python interpreter is to start the interpreter in interactive mode from the command line and run the command `import pygrametl`.

```
Python 3.9.2 (default, Feb 20 2021, 18:40:11)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pygrametl
>>>
```

If this fails with the message `ImportError: No module named pygrametl` then verify that the install location of the package is included in either the environment variable, `PYTHONPATH` if CPython is used, or the environment variable `JYTHONPATH` if Jython is used. By including the location of pygrametl in these variables, it is available to all instances of that Python implementation just like any built-in Python package. As an alternative, the path to pygrametl can be set on a program to program basis, by adding the path of pygrametl to `sys.path`, before importing the package in your code.

```
# The path to the pygrametl package is added to the path used by the Python
# interpreter when modules are being imported, this must be done in all
# program using a module not included in the default Python path
import sys
sys.path.append('/path/to/pygrametl')

# After the folder is added to Python's path can pygrametl be imported
import pygrametl
```

1.2 Beginner Guide

The following is a small guide for users new to pygrametl. It shows the main constructs provided by pygrametl and how to use them to create a simple ETL flow for a made-up example. The example is a toy data warehouse for a chain of book stores and is shown below as *Data Warehouse example*. The warehouse has one fact table and three dimensions organized in a star schema. The fact table stores facts about how many of each book is sold each day. The book dimension stores the name and genre of each book sold, the location dimension stores the city and region of the stores, and the time dimension stores the date of each sale. To keep the example simple, none of the dimensions are snowflaked, nor do they contain any slowly changing attributes. These are however supported by pygrametl through *SnowflakedDimension*, *TypeOneSlowlyChangingDimension*, and *SlowlyChangingDimension*, respectively. In addition, pygrametl provides high-level constructs for creating efficient multiprocess or multithreaded ETL flow, depending on the implementation of Python used (see *Parallel*). pygrametl also simplifies testing by making it easy to define preconditions and postconditions for each part of an ETL flow without relying on external files for the input and the expected results (see *Drawn Table Testing*).

Note: When using pygrametl, we strongly recommend using named parameters when instantiating classes as this improves readability and prevents errors in the future if the API changes.

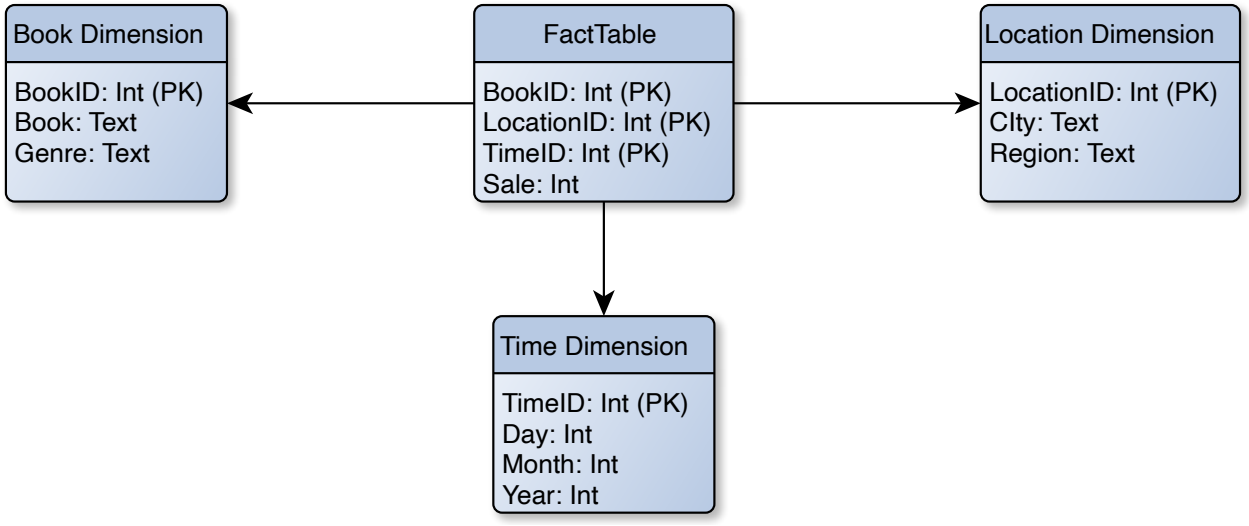

Fig. 1: Data Warehouse example

1.2.1 Input Data

Most pygrametl abstractions either produce, consume, or operate on data in rows. A row is a Python `dict` where the names of the row's columns are the keys and the values are the data the row contains. For more information about the data sources provided by pygrametl see *Data Sources*.

Note: The data and Python source code in this guide can be downloaded using the following link.

The most important data for the warehouse are which books have been sold. This information can be extracted from the book stores' sales records which are stored in the SQLite database sale.sqlite. Of course, storing sales records from multiple book stores is not a common use-case for SQLite. However, SQLite is used in this beginner guide as it makes sharing the input data simple and demonstrates pygrametl's ability to read and write from any RDBMS that provides a **PEP 249** connection. The data is stored in the table sale as shown below:

```
| book:text             | genre:text | store:text | date:date      | sale:int |
| --------------------- | ---------- | ---------- | -------------- | -------- |
| Nineteen Eighty-Four  | Novel      | Aalborg    | 2005/08/05     | 50       |
| Calvin and Hobbes One | Comic      | Aalborg    | 2005/08/05     | 25       |
| The Silver Spoon      | Cookbook   | Aalborg    | 2005/08/14     | 5        |
| The Silver Spoon      | Cookbook   | Odense     | 2005/09/01     | 7        |
| ....                  |            |            |                |          |
```

The book titles and genres are extracted from the CMU Book Summary Dataset. As the geographical information stored in the sales records is limited, the location dimension must be pre-filled with data from the CSV file region.csv. This file contains data about cities and regions as shown below (the tabs are added for readability):

```
city,       region
Aalborg,    North Denmark Region
Odense,     Region of Southern Denmark
....
```

1.2.2 ETL Flow

The ETL flow is designed to run on CPython and use PostgreSQL as the RDBMS for the data warehouse. The guide assumes PostgreSQL is already installed and running. The ETL flow can easily be run on other Python implementations like Jython. For example, to use Jython the **PEP 249** database drivers must simply be replaced with their JDBC equivalents and *ConnectionWrapper* with *JDBCConnectionWrapper*. For more information about running pygrametl on Jython see *Jython*.

We start by creating the database and tables for the data warehouse in PostgreSQL using psql. The SQL script example.sql creates the dw database, the dwuser role with all privileges, and the four tables:

```
psql -f example.sql
```

For the ETL flow we start by importing the various functions and classes needed in this beginner guide. The psycopg2 and sqlite3 database drivers must be imported so a connection to PostgreSQL and SQLite can be established. The main pygrametl module is also imported so a *ConnectionWrapper* can be created. pyggrametl's *datasources* module is imported so the sales records (*SQLSource*) and CSV file (CSVSource) can be read. Finally, classes for interacting with the fact table (*FactTable*) and the various dimensions (*CachedDimension*) are imported from *tables*.

```
# psycopg2 is a database driver allowing CPython to access PostgreSQL
import psycopg2

# sqlite3 is a database driver allowing CPython to access SQLite
import sqlite3

# pygrametl's __init__ file provides a set of helper functions and more
# importantly the class ConnectionWrapper for wrapping PEP 249 connections
import pygrametl

# pygrametl makes it simple to read external data through datasources
from pygrametl.datasources import SQLSource, CSVSource

# Interacting with the dimensions and the fact table is done through a set
# of classes. A suitable object must be created for each table
from pygrametl.tables import CachedDimension, FactTable
```

Then a connection to the database containing the sales records and the data warehouses is needed. For CPython, these must be **PEP 249** connections. As the data warehouse connection will be shared by multiple pygrametl abstractions, an instance of *ConnectionWrapper* is created. The first instance created of this class is set as the default connection for pygrametl's abstractions. This allows pygrametl to be used without having to pass a connection to each abstraction that needs it. A *ConnectionWrapper* is not needed for the connection to the sales database as it is only used by the CSVSource, so in that case, the **PEP 249** connection is used directly. For more information about database connections in pygrametl see *Database*.

```
# Creates a PEP 249 connection to the sales database. PARSE_DECLTYPES makes
# sqlite3 return values with the types specified in the database's schema
sale_conn = sqlite3.connect("sale.sqlite",
        detect_types=sqlite3.PARSE_DECLTYPES)

# While SQLite is used in this guide, any RDBMS that provides a PEP 249
# driver can be used with pygrametl. For example, SQLite can be replaced
# with PostgreSQL by simply replacing sale_conn with following two lines
# sale_string = "host='localhost' dbname='sale' user='user' password='pass'"
# sale_conn = psycopg2.connect(sale_string)
```

(continues on next page)

(continued from previous page)

```
# A connection is also created to the data warehouse. The connection is
# then given to a ConnectionWrapper so it becomes implicitly shared between
# all the pygrametl abstractions that needs it without being passed around
dw_string = "host='localhost' dbname='dw' user='dwuser' password='dwpass'"
dw_conn = psycopg2.connect(dw_string)

# Although the ConnectionWrapper is shared automatically between pygrametl
# abstractions, it is saved in a variable so the connection can be closed
dw_conn_wrapper = pygrametl.ConnectionWrapper(connection=dw_conn)
```

To get data into the ETL flow, two data sources are created. One for the database containing the sales records, and one for the CSV file containing the region information. For more information about the various data sources provided by pygrametl see *Data Sources*.

```
# The location dimension stores the name of a location in the column city
# instead of in the column store as done in the input data from the sales
# database. By passing SQLSource a sequence of names matching the number of
# columns in the table it can automatically rename the columns
name_mapping = 'book', 'genre', 'city', 'date', 'sale'

# Extraction of rows from a database using a PEP 249 connection and SQL
query = "SELECT book, genre, store, date, sale FROM sale"
sale_source = SQLSource(connection=sale_conn, query=query,
        names=name_mapping)

# Extraction of rows from a CSV file does not require a PEP 249 connection,
# just an open file handler. pygrametl uses Python's DictReader for CSV
# files and assumes the header of the CSV file contains the name of each
# column. When using CSVSource it is very important to convert the values
# to the correct type before inserting them into a table through pygrametl.
# As region.csv is encoded as UTF-8 and contains the non-ASCII characters æ
# and ø, the encoding open() will use is explicitly set to UTF-8. Of course,
# if a file uses a different encoding than UTF-8 it should be used instead
region_file_handle = open('region.csv', 'r', 16384, "utf-8")
region_source = CSVSource(f=region_file_handle, delimiter=',')
```

An object must then be created for each dimension and fact table in the data warehouse. pygrametl provides many types of abstractions for dimensions and fact tables, but in this example, we use the simplest ones. For more information about the more advanced dimension and fact table classes, see *Dimensions* and *Fact Tables*.

```
# An instance of CachedDimension is created for each dimension in the data
# warehouse. CachedDimension uses a local cache to significantly reduce the
# number of requests issued to the RDBMS. CachedDimension should generally
# be used instead of Dimension unless the higher memory consumption causes
# problems. For each dimension, the name of the database table, the table's
# primary key, and the table's non-key columns (attributes) are given. In
# addition, for the location dimension, the subset of the attributes that
# should be used to lookup the primary key is given. As mentioned in the
# beginning of this guide, using named parameters is strongly encouraged
book_dimension = CachedDimension(
        name='book',
```

(continues on next page)

(continued from previous page)

```
        key='bookid',
        attributes=['book', 'genre'])

time_dimension = CachedDimension(
        name='time',
        key='timeid',
        attributes=['day', 'month', 'year'])

location_dimension = CachedDimension(
        name='location',
        key='locationid',
        attributes=['city', 'region'],
        lookupatts=['city'])

# A single instance of FactTable is created for the data warehouse's single
# fact table. It is created with the name of the table, a list of columns
# constituting the primary key of the fact table, and a list of measures
fact_table = FactTable(
        name='facttable',
        keyrefs=['bookid', 'locationid', 'timeid'],
        measures=['sale'])
```

As the input dates are datetime objects and the time dimension consists of multiple levels (day, month, and year), the datetime objects must be split into their separate values. For this, a normal Python function is created and passed each of the rows. As pygrametl is a Python package, data transformations can be implemented using standard Python without any syntactic additions or restrictions. This also means that Python's many packages can be used as part of an ETL flow.

```
# A normal Python function is used to split the date into its parts
def split_date(row):
    """Splits a date represented by a datetime into its three parts"""

    # First the datetime object is extracted from the row dictionary
    date = row['date']

    # Then each part is reassigned to the row dictionary. It can then be
    # accessed by the caller as the row is a reference to the dict object
    row['year'] = date.year
    row['month'] = date.month
    row['day'] = date.day
```

Finally, the data can be inserted into the data warehouse. All rows from the CSV file are inserted into the location dimension first. This is necessary for foreign keys to the location dimension to be computed while filling the fact table. The other two dimensions are filled while inserting the facts as the data needed is included in the sales records. To ensure that the data is committed to the database and that the connection is closed correctly, the methods *ConnectionWrapper.commit()* and *ConnectionWrapper.close()* are executed at the end.

```
# The Location dimension is filled with data from the CSV file as the file
# contains all the information required for both columns in the table. If
# the dimension was filled using data from the sales database, it would be
# necessary to update the region attribute with data from the CSV file
# later. To insert the rows the method CachedDimension.insert() is used
```

(continues on next page)

(continued from previous page)

```
[location_dimension.insert(row) for row in region_source]

# The file handle to the CSV file can then be closed
region_file_handle.close()

# All the information needed for the other dimensions are stored in the
# sales database. So with only a single iteration over the sales records
# the ETL flow can split the date and lookup the three dimension keys
# needed for the fact table. While retrieving the dimension keys, pygrametl
# can automatically update the dimensions with new data if ensure() is
# used. This method combines a lookup with an insertion so a new row is
# only inserted into the dimension or fact table if it does not yet exist
for row in sale_source:

    # The date is split into its three parts
    split_date(row)

    # The row is updated with the correct primary keys for each dimension, and
    # any new data are inserted into each of the dimensions at the same time
    row['bookid'] = book_dimension.ensure(row)
    row['timeid'] = time_dimension.ensure(row)

    # CachedDimension.ensure() is not used for the location dimension as it
    # has already been filled. Instead the method CachedDimension.lookup()
    # is used. CachedDimension.lookup() does not insert any data and
    # returns None if a row with the correct lookupatts is not available.
    # This makes error handling very simple to implement. In this case an
    # error is raised if a location is missing from the CSV file as
    # recovery is not possible
    row['locationid'] = location_dimension.lookup(row)
    if not row['locationid']:
        raise ValueError("city was not present in the location dimension")

    # As the number of sales is already aggregated in the sales records, the
    # row can now be inserted into the data warehouse. If aggregation, or
    # other more advanced transformations are required, the full power of
    # Python is available as shown with the call to split_date()
    fact_table.insert(row)

# After all the data have been inserted, the connection is ordered to
# commit and is then closed. This ensures that the data is committed to the
# database and that the resources used by the connection are released
dw_conn_wrapper.commit()
dw_conn_wrapper.close()

# Finally, the connection to the sales database is closed
sale_conn.close()
```

This small example shows how to quickly create a very simple ETL flow with pygrametl. A combined version with fewer comments can be seen below. However, since this is a very small and simple example, the batching and bulk loading built into some of the more advanced dimension and fact table classes has not been used. For larger ETL flows, these can be used to significantly increase the throughput of an ETL flow. See *Dimensions* and *Fact Tables* for more information. The simple parallel capabilities of pygrametl can also be used to further increase the throughput of an

ETL flow (see *Parallel*), and the correctness of an ETL flow should also be checked using a set of automated repeatable tests (see *Drawn Table Testing*).

```
import psycopg2
import sqlite3
import pygrametl
from pygrametl.datasources import SQLSource, CSVSource
from pygrametl.tables import CachedDimension, FactTable

# Opening of connections and creation of a ConnectionWrapper
sale_conn = sqlite3.connect("sale.sqlite",
        detect_types=sqlite3.PARSE_DECLTYPES)

dw_string = "host='localhost' dbname='dw' user='dwuser' password='dwpass'"
dw_conn = psycopg2.connect(dw_string)
dw_conn_wrapper = pygrametl.ConnectionWrapper(connection=dw_conn)

# Creation of data sources for the sales database and the CSV file
# containing extra information about cities and regions in Denmark
name_mapping = 'book', 'genre', 'city', 'date', 'sale'
query = "SELECT book, genre, store, date, sale FROM sale"
sale_source = SQLSource(connection=sale_conn, query=query,
        names=name_mapping)

region_file_handle = open('region.csv', 'r', 16384, "utf-8")
region_source = CSVSource(f=region_file_handle, delimiter=',')

# Creation of dimension and fact table abstractions for use in the ETL flow
book_dimension = CachedDimension(
        name='book',
        key='bookid',
        attributes=['book', 'genre'])

time_dimension = CachedDimension(
        name='time',
        key='timeid',
        attributes=['day', 'month', 'year'])

location_dimension = CachedDimension(
        name='location',
        key='locationid',
        attributes=['city', 'region'],
        lookupatts=['city'])

fact_table = FactTable(
        name='facttable',
        keyrefs=['bookid', 'locationid', 'timeid'],
        measures=['sale'])

# Python function needed to split the date into its three parts
def split_date(row):
    """Splits a date represented by a datetime into its three parts"""
```

(continues on next page)

(continued from previous page)

```
    # Splitting of the date into parts
    date = row['date']
    row['year'] = date.year
    row['month'] = date.month
    row['day'] = date.day

# The location dimension is loaded from the CSV file
[location_dimension.insert(row) for row in region_source]

# The file handle for the CSV file can then be closed
region_file_handle.close()

# Each row in the sales database is iterated through and inserted
for row in sale_source:

    # Each row is passed to the date split function for splitting
    split_date(row)

    # Lookups are performed to find the key in each dimension for the fact
    # and if the data is not there, it is inserted from the sales row
    row['bookid'] = book_dimension.ensure(row)
    row['timeid'] = time_dimension.ensure(row)

    # The location dimension is pre-filled, so a missing row is an error
    row['locationid'] = location_dimension.lookup(row)
    if not row['locationid']:
        raise ValueError("city was not present in the location dimension")

    # The row can then be inserted into the fact table
    fact_table.insert(row)

# The data warehouse connection is then ordered to commit and close
dw_conn_wrapper.commit()
dw_conn_wrapper.close()

# Finally, the connection to the sales database is closed
sale_conn.close()
```

CHAPTER

TWO

CODE EXAMPLES

2.1 Database

Database access in pygrametl is done through either a **PEP 249** connection if CPython is used, or with a JDBC connection when pygrametl is running on Jython. pygrametl provides multiple abstractions on top of these connections and direct usage of these to manipulate the database should generally not be necessary. As an abstraction for database rows Python's `dict` type is used, where the keys the names of the columns in the table and the values are the data stored in that row.

2.1.1 Connection Wrappers

Multiple connection wrappers are provided by the pygrametl framework to allow **PEP 249** connections and JDBC connections to be used uniformly, and to allow multiple threads and process to use the connection safely. In addition, the connection wrappers for **PEP 249** connections also automatically convert from the pyformat parameter style used by pygrametl to any of the other parameter styles defined in **PEP 249#paramstyle**. To simplify the use of database connections, the first connection wrapper created is set as the default. The default connection wrapper can be used by abstractions such as *tables.FactTable* and *tables.Dimension* without the user having to pass the connection wrapper to them explicitly. If another database connection should be used, for example, if data is read from one database and written to another, a specific connection can be explicitly passed as an argument to all pygrametl abstractions that can read to and/or write from a database.

ConnectionWrapper and *JDBCConnectionWrapper.JDBCConnectionWrapper* are the two main connection wrappers provided by pygrametl. The interface provided by these two classes is just an abstraction on top of database operations, and provides methods, among others, for executing statements, iterating over returned rows, and committing transactions. Note however that these connection wrappers cannot be used by multiple threads or processes in parallel. To ensure that database access is performed correctly in a parallel ETL program without burdening the user with the task, the class `parallel.SharedConnectionWrapperClient` is provided. This class can be created from an existing connection wrapper using the function *parallel.shareconnectionwrapper()*. Each separate process can then be given a unique copy of the shared connection to access the database safely in parallel. For more information about the parallel capabilities of pygrametl see *Parallel*.

2.1.2 Experimental Connection Wrappers

pygrametl also provides two very experimental connection wrappers: `BackgroundConnectionWrapper` and *JDBCConnectionWrapper.BackgroundJDBCConnectionWrapper*. They are provided as alternatives to *ConnectionWrapper* and *JDBCConnectionWrapper.JDBCConnectionWrapper* and perform the database operations in a separate thread instead of the same thread as the ETL program. As they are considered experimental, they are not set as default upon creation, and must thus manually be set as the default with the method `setasdefault()`, available on all connection wrappers, or be manually passed around the program.

For most usage the classes *ConnectionWrapper* and *JDBCConnectionWrapper.JDBCConnectionWrapper* will likely provide better performance compared to the background versions. Furthermore, a connection wrapper used in a parallel ETL program should always be wrapped using *parallel.shareconnectionwrapper()* to ensure safe parallel database access, which itself runs the connection wrapper in a separate process or thread depending on the implementation. As the two implementations are very similar and provide an identical interface, either set of implementations might be removed in a future release.

2.2 Data Sources

pygrametl supports numerous data sources, which are iterable classes that produce rows. A row is a Python `dict` where the keys are the names of the columns in the table where the row is from, and the values are the data stored in that row. Users can easily implement new data sources by implementing a version of the `__iter__()` method that returns `dict`. As data source are iterable, they can, e.g., be used in a loop as shown below:

```
for row in datasource:
    ...
```

While users can define their own data sources, pygrametl includes a number of commonly used data sources:

2.2.1 SQLSource

SQLSource is a data source used to iterate over the results of a single SQL query. The data source's constructor must be passed a **PEP 249** connection and not a *ConnectionWrapper*. As an example, a PostgreSQL connection created using the psycopg2 package is used below:

```
import psycopg2
from pygrametl.datasources import SQLSource

conn = psycopg2.connect(database='db', user='dbuser', password='dbpass')
sqlSource = SQLSource(connection=conn, query='SELECT * FROM table')
```

In the above example, an *SQLSource* is created in order to extract all rows from the table named table.

A tuple of strings can also optionally be supplied as the parameter `names`, to automatically rename the elements in the query results. Naturally, the number of supplied names must match the number of elements in the result:

```
import psycopg2
from pygrametl.datasources import SQLSource

conn = psycopg2.connect(database='db', user='dbuser', password='dbpass')
sqlSource = SQLSource(connection=conn, query='SELECT * FROM table',
                      names=('id', 'name', 'price'))
```

SQLSource also makes it possible to supply an SQL expression that will be executed before the query, through the `initsql` parameter. The result of the expression will not be returned. In the example below a new view is created and then used in the query:

```
import psycopg2
from pygrametl.datasources import SQLSource

conn = psycopg2.connect(database='db', user='dbuser', password='dbpass')
sqlSource = SQLSource(connection=conn, query='SELECT * FROM view',
    initsql='CREATE VIEW view AS SELECT id, name FROM table WHERE price > 10')
```

2.2.2 CSVSource

`CSVSource` is a data source that returns a row for each line in a character-separated file. It is an alias for Python's csv.DictReader as it already is iterable and returns `dict`. An example of how to use `CSVSource` to read a file containing comma-separated values is shown below:

```
from pygrametl.datasources import CSVSource

# ResultsFile.csv contains: name,age,score
csvSource = CSVSource(f=open('ResultsFile.csv', 'r', 16384), delimiter=',')
```

In the above example, a `CSVSource` is initialized with a file handler that uses a buffer size of 16384, This particular buffer size is used as it performed better than the alternatives we evaluated it against.

2.2.3 TypedCSVSource

TypedCSVSource extends `CSVSource` with typecasting by wrapping csv.DictReader instead of simply being an alias.

```
from pygrametl.datasources import TypedCSVSource

# ResultsFile.csv contains: name,age,score
typedCSVSource = TypedCSVSource(f=open('ResultsFile.csv', 'r', 16384),
                                casts={'age': int, 'score': float},
                                delimiter=',')
```

In the above example, a *TypedCSVSource* is initialized with a file handler that uses a buffer size of 16384. This particular buffer size is used as it performed better than the alternatives we evaluated it against. A dictionary is also passed which provides information about what type each column should be cast to. A cast is not performed for the name column as *TypedCSVSource* uses `str` as the default.

2.2.4 PandasSource

PandasSource wraps a Pandas DataFrame so it can be used as a data source. The class reuses existing functionality provided by DataFrame. An example of how to use this class can be seen below. In this example data is loaded from a spreadsheet, then transformed using a Pandas DataFrame, and last converted to an iterable that produce `dict` for use with pygrametl:

```
import pandas
from pygrametl.datasources import PandasSource

```

(continues on next page)

(continued from previous page)

```
df = pandas.read_excel('Revenue.xls')
df['price'] = df['price'].apply(lambda p: float(p) / 7.46)
pandasSource = PandasSource(df)
```

In the above example, a Pandas DataFrame is created from a spreadsheet containing revenue from some form of sales. Afterwards the data of the price column is transformed using one of the higher-order functions built into the Pandas package. Last, so the data can be loaded into a data warehouse using pygrametl, a *PandasSource* is created with the DataFrame as an argument, making the rows of the DataFrame accessible through a data source.

2.2.5 MergeJoiningSource

In addition to the above data sources which reads data from external sources, pygrametl also includes a number of data sources that take other data sources as input to transform and/or combine them.

MergeJoiningSource can be used to equijoin the rows from two data sources. The rows of the two data sources must be delivered in sorted order. The shared attributes on which the rows are to be joined must also be given.

```
from pygrametl.datasources import CSVSource, MergeJoiningSource

products = CSVSource(f=open('products.csv', 'r', 16384), delimiter=',')
sales = CSVSource(f=open('sales.csv', 'r', 16384), delimiter='\t')
mergeJoiningSource = MergeJoiningSource(src1=products, key1='productid',
                                        src2=sales, key2='productid')
```

In the above example, a *MergeJoiningSource* is used to join two data sources on their shared attribute productid.

2.2.6 HashJoiningSource

HashJoiningSource functions similarly to *MergeJoiningSource*, but it performs the join using a hash map. Thus the two input data sources need not produce their rows in sorted order.

```
from pygrametl.datasources import CSVSource, HashJoiningSource

products = CSVSource(f=open('products.csv', 'r', 16384), delimiter=',')
sales = CSVSource(f=open('sales.csv', 'r', 16384), delimiter='\t')
hashJoiningSource = HashJoiningSource(src1=products, key1='productid',
                                      src2=sales, key2='productid')
```

2.2.7 UnionSource

The class *UnionSource* creates a union of a number of the supplied data sources. *UnionSource* does not require that the input data sources all produce rows containing the same attributes, which also means that an *UnionSource* does not guarantee that all of the rows it produces contain the same attributes.

```
from pygrametl.datasources import CSVSource, UnionSource

salesOne = CSVSource(f=open('sales1.csv', 'r', 16384), delimiter='\t')
salesTwo = CSVSource(f=open('sales2.csv', 'r', 16384), delimiter='\t')
salesThree = CSVSource(f=open('sales3.csv', 'r', 16384), delimiter='\t')
```

(continues on next page)

(continued from previous page)

```
combinedSales = UnionSource(salesOne, salesTwo, salesThree)
```

Each data source are exhausted before the next data source is read. This means that all rows are read from the first data source before any rows are read from the second data source, and so on.

2.2.8 RoundRobinSource

It can also be beneficial to interleave rows, and for this purpose, *RoundRobinSource* can be used.

```
from pygrametl.datasources import CSVSource, RoundRobinSource

salesOne = CSVSource(f=open('sales1.csv', 'r', 16384), delimiter='\t')
salesTwo = CSVSource(f=open('sales2.csv', 'r', 16384), delimiter='\t')
salesThree = CSVSource(f=open('sales3.csv', 'r', 16384), delimiter='\t')

combinedSales = RoundRobinSource((salesOne, salesTwo, salesThree),
                                 batchsize=500)
```

In the above example, *RoundRobinSource* is given a number of data sources, and the argument `batchsize`, which are the number of rows to be read from one data source before reading from the next in a round-robin fashion.

2.2.9 ProcessSource

ProcessSource is used for iterating over a data source using a separate worker process or thread. The worker reads data from the input data source and creates batches of rows. When a batch is complete, it is added to a queue so it can be consumed by another process or thread. If the queue is full the worker blocks until an element is removed from the queue. The sizes of the batches and the queue are optional parameters, but tuning them can often improve throughput. For more examples of the parallel features provided by pygrametl see *Parallel*.

```
from pygrametl.datasources import CSVSource, ProcessSource

sales = CSVSource(f=open('sales.csv', 'r', 16384), delimiter='\t')
processSource = ProcessSource(source=sales, batchsize=1000, queuesize=20)
```

2.2.10 FilteringSource

FilteringSource is used to apply a filter to a data source. By default, the built-in Python function bool is used, which can be used to remove empty rows. Alternatively, the user can supply a custom filter function, which should be a callable function `f(row)`, which returns `True` when a row should be passed on. In the example below, rows are removed if the value of their location attribute is not Aalborg.

```
from pygrametl.datasources import CSVSource, FilteringSource


def locationfilter(row):
    row['location'] == 'Aalborg'
```

(continues on next page)

(continued from previous page)

```
sales = CSVSource(f=open('sales.csv', 'r', 16384), delimiter='\t')
salesFiltered = FilteringSource(source=sales, filter=locationfilter)
```

2.2.11 MappingSource

MappingSource can be used to apply functions to the columns of a data source. It can be given a dictionary that where the keys are the columns and the values are callable functions of the form `f(val)`. The functions will be applied to the attributes in an undefined order. In the example below, a function is used to cast all values for the attribute price to integers while rows are being read from a CSV file.

```
from pygrametl.datasources import CSVSource, MappingSource

sales = CSVSource(f=open('sales.csv', 'r', 16384), delimiter=',')
salesMapped = MappingSource(source=sales, callables={'price': int})
```

2.2.12 TransformingSource

TransformingSource can be used to apply functions to the rows of a data source. The class can be supplied with a number of callable functions of the form `f(row)`, which will be applied to the source in the given order.

```
import pygrametl
from pygrametl.datasources import CSVSource, TransformingSource


def dkk_to_eur(row):
    price_as_a_number = int(row['price'])
    row['dkk'] = price_as_a_number
    row['eur'] = price_as_a_number / 7.43


sales = CSVSource(f=open('sales.csv', 'r', 16384), delimiter=',')
salesTransformed = TransformingSource(sales, dkk_to_eur)
```

In the above example, the price is converted from a string to an integer and stored in the row as two currencies.

2.2.13 CrossTabbingSource

CrossTabbingSource can be used to compute the cross tab of a data source. The class takes as parameters the names of the attributes that are to appear as rows and columns in the crosstab, as well as the name of the attribute to aggregate. By default, the values are aggregated using *pygrametl.aggregators.Sum*, but the class also accepts an alternate aggregator from the module *pygrametl.aggregators*.

```
from pygrametl.datasources import CSVSource, CrossTabbingSource, \
    TransformingSource
from pygrametl.aggregators import Avg


def dkk_to_eur(row):
    price_as_a_number = int(row['price'])
```

(continues on next page)

(continued from previous page)

```
    row['dkk'] = price_as_a_number
    row['eur'] = price_as_a_number / 7.43


sales = CSVSource(f=open('sales.csv', 'r', 16384), delimiter=',')
salesTransformed = TransformingSource(sales, dkk_to_eur)

crossTab = CrossTabbingSource(source=salesTransformed, rowvaluesatt='product',
                              colvaluesatt='location', values='eur',
                              aggregator=Avg())
```

In the above example, a crosstab is made from a table containing sales data in order to view the average price of products across different locations. *TransformingSource* is used to parse and convert the price from DKK to EUR.

2.2.14 DynamicForEachSource

DynamicForEachSource is a data source that for each data source provided as input, creates a new data source that will be iterated by the *DynamicForEachSource* data source. To create the new data sources the user must provide a function that when called with a single argument, return a new data source. In the example below, *DynamicForEachSource* is used to create a CSVSource for each of the CSV files in a directory. The *DynamicForEachSource* stores the input list in a safe multiprocessing queue, and as such the *DynamicForEachSource* instance can be given to several *ProcessSource*. For information about pygrametl's parallel features see *Parallel*.

```
import glob
from pygrametl.datasources import CSVSource, DynamicForEachSource


def createCSVSource(filename):
    return CSVSource(f=open(filename, 'r', 16384), delimiter=',')


salesFiles = glob.glob('sales/*.csv')
combinedSales = DynamicForEachSource(seq=salesFiles, callee=createCSVSource)
```

2.3 Dimensions

Multiple abstractions for representing dimensions in a data warehouse are provided by pygrametl. These abstractions allow for simple modeling of both star and snowflake schemas as well as type 1, type 2, and combined type 1 and type 2 slowly changing dimensions. The abstractions can be used for both sequential and parallel loading of data into dimensions. For more information about the parallel capabilities of pygrametl see *Parallel*. The following examples use PostgreSQL as the RDBMS and psycopg2 as the database driver.

All of following classes are currently implemented in the *pygrametl.tables* module.

2.3.1 Dimension

Dimension is the simplest abstraction pygrametl provides for interacting with dimensions in a data warehouse. It provides an interface for performing operations on the underlying table, such as insertions or looking up keys, while abstracting away the database connection and queries. Using *Dimension* is a two-step process. Firstly, an instance of *ConnectionWrapper* must be created with a **PEP 249** database connection. The first *ConnectionWrapper* created is automatically set as the default and used by *Dimension* for interacting with the database. For more information about database connections see *Database*. Secondly, an instance of *Dimension* must be created for each dimension in the data warehouse. To create an instance of *Dimension* the name of the table must be specified, along with the primary key of the table, as well as the non-key columns in the table. In addition to these required parameters, a subset of columns to be used for looking up keys can also be specified, as well as a function for computing the primary key, a default return value if a lookup fails, and a function for expanding a row automatically.

```
import psycopg2
import pygrametl
from pygrametl.tables import Dimension

# Input is a list of rows which in pygrametl is modeled as dicts
products = [
    {'name': 'Calvin and Hobbes 1', 'category': 'Comic', 'price': '10'},
    {'name': 'Calvin and Hobbes 2', 'category': 'Comic', 'price': '10'},
    {'name': 'Calvin and Hobbes 3', 'category': 'Comic', 'price': '10'},
    {'name': 'Cake and Me', 'category': 'Cookbook', 'price': '15'},
    {'name': 'French Cooking', 'category': 'Cookbook', 'price': '50'},
    {'name': 'Sushi', 'category': 'Cookbook', 'price': '30'},
    {'name': 'Nineteen Eighty-Four', 'category': 'Novel', 'price': '15'},
    {'name': 'The Lord of the Rings', 'category': 'Novel', 'price': '60'}
]

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)

# The instance of Dimension connects to the table product in the
# database using the default connection wrapper created above, the
# argument lookupatts specifies the column which needs to match
# when doing a lookup for the key from this dimension
productDimension = Dimension(
    name='product',
    key='productid',
    attributes=['name', 'category', 'price'],
    lookupatts=['name'])

# Filling a dimension is simply done by using the insert method
for row in products:
    productDimension.insert(row)

# Ensures that the data is committed and the connection is closed correctly
```

(continues on next page)

(continued from previous page)

```
conn.commit()
conn.close()
```

In the above example, a set of rows with product information is loaded into the product dimension, using an instance of *Dimension* created with information about the table in the database. The list of product information can then be inserted into the database one element at a time using the method *Dimension.insert()*. Then the transaction must be committed and the connection closed to ensure that the data is written to the database.

2.3.2 CachedDimension

CachedDimension extends *Dimension* with a cache to reduce the latency of lookups by reducing the number of round trips to the database. To control what is cached, three additional parameters have been added to its constructor. The parameter `prefill` indicates that the cache should be filled with data from the database when the object is created, while `cachefullrows` determines whether only the primary key and columns defined by `lookuparts`, or entire rows should be cached. Lastly, the parameter `cacheoninsert` specifies if newly inserted rows should be cached. To ensure that the cache is kept consistent, the RDBMS is not allowed to modify the rows in any way, a default value set by the RDBMS is an example of a simple-to-miss violation of this.

```
import psycopg2
import pygrametl
from pygrametl.datasources import CSVSource
from pygrametl.tables import CachedDimension, FactTable

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(pgconn)

# The cached dimension is initialized with data from the product table in
# the database, allowing for more efficient lookups of keys for the fact
# table, at the cost of requiring it to already contain the necessary data
productDimension = CachedDimension(
    name='product',
    key='productid',
    attributes=['name', 'category', 'price'],
    lookupatts=['name'],
    prefill=True)

# A similar abstraction is created for the data warehouse fact table
factTable = FactTable(
    name='facttable',
    measures=['sales'],
    keyrefs=['storeid', 'productid', 'dateid'])

# A CSV file contains information about each product sold by a store
sales = CSVSource(f=open('sales.csv', 'r', 16384), delimiter='\t')
```

(continues on next page)

(continued from previous page)

```
# Looking up keys from the product dimension is done using the lookup
# method with the information read from the sales.csv file. The second
# argument renames the column product_name from the CSV file to name
for row in sales:

    # Using only the attributes defined as lookupatts, lookup() checks if a
    # row with matching values are present in the cache. Only if a match
    # cannot be found in the cache does lookup() check the database table.
    row['productid'] = productDimension.lookup(row, {'name': 'product_name'})
    factTable.insert(row)

# Ensures that the data is committed and the connection is closed correctly
conn.commit()
conn.close()
```

The example shows how to utilize *CachedDimension* to automatically improve the performance of *CachedDimension.lookup()*. The *CachedDimension* caches the values from the product dimension locally, allowing increased performance when looking up keys as fewer, or none if all rows are cached, round trips are made to the database.

2.3.3 BulkDimension

BulkDimension is a dimension optimized for high throughput when inserting rows and fast lookups. This is done by inserting rows in bulk from a file while using an in-memory cache for lookup. To support this the RDBMS is not allowed to modify the rows in any way, as this would make the cache and the database table inconsistent. Another aspect of *BulkDimension* is that `BulkDimension.update()` and `BulkDimension.getbyvals()` calls `BulkDimension.endload()` which inserts all rows stored in the file into the database using a user-defined bulk loading function. Thus, calling these functions often will negate the benefit of bulk loading. The method *BulkDimension.getbykey()* also forces *BulkDimension* to bulk load by default but can use the cache if `cachefullrows` is enabled at the cost of additional memory. `BulkDimension.lookup()` and `BulkDimension.ensure()` will always use the cache and do not invoke any database operations as *BulkDimension* never evicts rows from its cache. If the dimension is too large to be cached in memory, the class *CachedBulkDimension* should be used instead as it supports bulk loading using a finite cache. To support bulk loading from a file on disk, multiple additional parameters have been added to *BulkDimension* constructor. These provide control over the temporary file used to store rows, such as specific delimiters and the number of rows to be bulk loaded. All of these parameters have a default value except for `bulkloader`. This parameter must be passed a function to be called for each set of rows to be bulk loaded, this is necessary as the exact way to perform bulk loading differs from RDBMS to RDBMS.

`func(name, attributes, fieldsep, rowsep, nullval, filehandle):`

Required signature of a function bulk loading data from a file into an RDBMS in pygrametl. For more information about bulk loading see *Bulk Loading*.

Arguments:

- name: the name of the dimension table in the data warehouse.
- attributes: a list containing the sequence of attributes in the dimension table.
- fieldsep: the string used to separate fields in the temporary file.
- rowsep: the string used to separate rows in the temporary file.
- nullval: if the *BulkDimension* was passed a string to substitute None values with, then it will be passed, if not then None is passed.

- filehandle: either the name of the file or the file object itself, depending upon on the value of `BulkDimension.usefilename`. Using the filename is necessary if the bulk loading is invoked through SQL (instead of directly via a method on the PEP249 driver). It is also necessary if the bulkloader runs in another process.

```
import sqlite3
import psycopg2
import pygrametl
from pygrametl.datasources import SQLSource
from pygrametl.tables import BulkDimension

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)


# This function bulk loads a file into PostgreSQL using psycopg2
def pgbulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):
    cursor = conn.cursor()
    # psycopg2 does not accept the default value used to represent NULL
    # by BulkDimension, which is None. Here this is ignored as we have no
    # NULL values that we wish to substitute for a more descriptive value
    cursor.copy_from(file=filehandle, table=name, sep=fieldsep,
                     columns=attributes)


# In addition to arguments needed for a Dimension, a reference to the
# bulk loader defined above must also be passed to BulkDimension
productDimension = BulkDimension(
    name='product',
    key='productid',
    attributes=['name', 'category', 'price'],
    lookupatts=['name'],
    bulkloader=pgbulkloader)

# A PEP249 connection is sufficient for an SQLSource so we do not need
# to create a new instance of ConnectionWrapper to read from the database
sqconn = sqlite3.connect('product_catalog.db')

# Encapsulating a database query in an SQLSource allows it to be used as an
# normal iterator, making it very simple to load the data into another table
sqlSource = SQLSource(connection=sqconn, query='SELECT * FROM product')

# Inserting data from a data source into a BulkDimension is performed just
# like any other dimension type in pygrametl, as the interface is the same
for row in sqlSource:
    productDimension.insert(row)
```

(continues on next page)

(continued from previous page)

```
# Ensures that the data is committed and the connections are closed correctly
conn.commit()
conn.close()
sqconn.close()
```

The example above shows how to use *BulkDimension* to efficiently load the contents of a local SQLite database into a data warehouse dimension. This process is a good use case for *BulkDimension* as `BulkDimension.update()`, *BulkDimension.getbykey()* and `BulkDimension.getbyval()` are not used, so no additional calls to `BulkDimension.endload()` are made. By bulk loading the rows from a file using `copy_from()` instead of inserting them one at a time, the time required to load the dimension is significantly reduced. However, it is important that *ConnectionWrapper.commit()* is executed after all rows have been inserted into *BulkDimension* as it ensures that the last set of rows are bulk loaded by calling `BulkDimension.endload()` on all tables. A downside of *BulkDimension* is that it caches the entire dimension in memory. If the dimension can be bulk loaded but is too large to cache in memory *CachedBulkDimension* should be used instead of *BulkDimension*.

2.3.4 CachedBulkDimension

CachedBulkDimension is very similar to *BulkDimension* and is also intended for bulk loading a dimension, so only their differences are described here. Unlike *BulkDimension* the size of *CachedBulkDimension* cache is limited by the parameter `cachesize`. This allows it to be used with a dataset too large to be cached entirely in memory. The trade-off is that *CachedBulkDimension.lookup()* and `CachedBulkDimension.ensure()` sometimes have to lookup keys in the database instead of always using the cache. However, the method *CachedBulkDimension.getbykey()* also no longer needs to call `CachedBulkDimension.endload()` if `cachefullrows` is not enabled. This is because *CachedBulkDimension* caches the rows currently in the file in a separate cache. All rows in the file are cached as there is no guarantee that the cache storing rows from the database does not evict rows currently in the file but not yet in the database when the cache is full, So an additional cache is needed to ensure that *CachedBulkDimension.lookup()* and *CachedBulkDimension.getbykey()* can locate rows before they are loaded into the database. *CachedBulkDimension.insert()* caches rows in the file cache, and only when the rows in the file are loaded into the database are they moved to the database row cache, in which *CachedBulkDimension.lookup()* also stores rows if the method had to query the database for them.

Due to the use of two caches, caching in *CachedBulkDimension* is controlled by two parameters. The parameter `cachesize` can be set to control the size of the cache for rows loaded into the database, while the parameter `bulksize` controls the number of rows stored in the file before the dimension bulk loads. As the rows in the file are all cached in a separate cache, the memory consumption will change in correspondence to both of these values.

Note: If rows with matching `lookupatts` are passed to `insert()` without bulk loading occurring between the calls, only the first row will be loaded into the dimension. All calls to `insert()` after the first one will just return the key of the first row as it is stored in the file cache.

2.3.5 TypeOneSlowlyChanging Dimension

TypeOneSlowlyChangingDimension allows the creation of a type 1 slowly changing dimension. The dimension is based on *CachedDimension*, albeit with a few differences. The primary difference between the two classes besides the additional method *TypeOneSlowlyChangingDimension.scdensure()*, is that *TypeOneSlowlyChangingDimension* always caches rows when they are inserted. This is done to minimize the number of round trips to the database needed for *TypeOneSlowlyChangingDimension.scdensure()* to increase its throughput. The user must also specify the sequence of attributes to use for looking up keys as `lookupatts`, and can optionally specify the sequence of type 1 slowly changing attributes `type1atts`. If `type1atts` is not given it will default to all attributes minus `lookupatts`. The sequences `lookupatts` and `type1atts` must be disjoint and an error

will be raised if they are not. As caching is used to increase to speedup lookups, it is assumed that the database does not change or add any attribute values to the rows. For example, a default value set by RDBMS and automatic type coercion can break this assumption.

```
import psycopg2
import pygrametl
from pygrametl.tables import TypeOneSlowlyChangingDimension

# Input is a list of rows which in pygrametl is modeled as dicts
products = [
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '10'},
    {'name': 'Cake and Me', 'category': 'Cookbook', 'price': '15'},
    {'name': 'French Cooking', 'category': 'Cookbook', 'price': '50'},
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '20'},
    {'name': 'Sushi', 'category': 'Cookbook', 'price': '30'},
    {'name': 'Nineteen Eighty-Four', 'category': 'Novel', 'price': '15'},
    {'name': 'The Lord of the Rings', 'category': 'Novel', 'price': '60'},
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '10'}
]

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)

# TypeOneSlowlyChangingDimension is created with price as a changing attribute
productDimension = TypeOneSlowlyChangingDimension(
    name='product',
    key='productid',
    attributes=['name', 'category', 'price'],
    lookupatts=['name'],
    type1atts=['price'])

# scdensure determines whether the row already exists in the database
# and inserts a new row or updates any type1atts that have changed
for row in products:
    productDimension.scdensure(row)

# Ensures that the data is committed and the connections are closed correctly
conn.commit()
conn.close()
```

The above example shows a scenario where the price of a product changes over time. The instance of *TypeOneSlowlyChangingDimension* automatically checks if a product already exists, and if it does, updates the price if the old and new values differ. As a type 1 slowly changing dimension does store the history of the changes only the value of the last row to be inserted will be stored, so the rows most be loaded in chronological order. If the history of the changes must be stored a type 2 slowly changing dimension should be created, and *SlowlyChangingDimension* should be used instead of *TypeOneSlowlyChangingDimension*.

2.3.6 SlowlyChangingDimension

SlowlyChangingDimension allows for the creation of either a type 2 slowly changing dimension, or a combined type 1 and type 2 slowly changing dimension. To support this functionality, multiple additional attributes have been added to *SlowlyChangingDimension* compared to *Dimension*. However, only the additional parameter `versionatt` is required when creating a *SlowlyChangingDimension*. This parameter indicates which of the dimensions attribute stores the row's version number. The method *SlowlyChangingDimension.scdensure()* updates the table while taking the slowly changing aspect of the dimension into account. If the row is already available then the primary key is returned, if the row is not available then it is inserted into the dimension, and if an attributes have changed a new version is created. The method *SlowlyChangingDimension.lookup()* is also changed slightly as it returns the latest version of a row. To improve the performance of lookups for a slowly changing dimension, caching is used, which assumes that the database does not modify any values in the inserted rows; an assumption that the use of default values can break.

```
import psycopg2
import pygrametl
from pygrametl.tables import SlowlyChangingDimension

# Input is a list of rows which in pygrametl is modeled as dicts
products = [
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '20',
     'date': '1990-10-01'},
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '10',
     'date': '1990-12-10'},
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '20',
     'date': '1991-02-01'},
    {'name': 'Cake and Me', 'category': 'Cookbook', 'price': '15',
     'date': '1990-05-01'},
    {'name': 'French Cooking', 'category': 'Cookbook', 'price': '50',
     'date': '1990-05-01'},
    {'name': 'Sushi', 'category': 'Cookbook', 'price': '30',
     'date': '1990-05-01'},
    {'name': 'Nineteen Eighty-Four', 'category': 'Novel', 'price': '15',
     'date': '1990-05-01'},
    {'name': 'The Lord of the Rings', 'category': 'Novel', 'price': '60',
     'date': '1990-05-01'}
]

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)

# This slowly changing dimension is created as type 2 only. Meaning that a
# new row only changes the validto attribute in the previous row. validto
# is a timestamp indicating when the row is no longer valid. As additional
# parameters, the object is initialized with information about which
# attribute holds a timestamp for when the row's validity starts and ends.
# The parameter fromfinder is also given, which is set to a function that
# computes the timestamp for when the row becomes valid. In this example,
```

(continues on next page)

(continued from previous page)

```
# the function datareader from pygrametl is used which converts a timestamp
# from a str to a datetime.date which PostgresSQL stores as Date type.
productDimension = SlowlyChangingDimension(
    name='product',
    key='productid',
    attributes=['name', 'category', 'price', 'validfrom', 'validto',
                'version'],
    lookupatts=['name'],
    fromatt='validfrom',
    fromfinder=pygrametl.datereader('date'),
    toatt='validto',
    versionatt='version')

# scdensure extends the ensure methods with support for updating slowly
# changing attributes of rows where lookupparts match. This is done by
# increamenting the version attribute for the new row, and assigning the new
# rows fromatt to the old rows toatt, indicating that the old row is no
# longer valid.
for row in products:
    productDimension.scdensure(row)

# Ensures that the data is committed and the connections are closed correctly
conn.commit()
conn.close()
```

As the values of the product dimension, in this case, have changing prices, a *SlowlyChangingDimension* is used to automate the changes a new row might incur on an existing row. The product information itself is also extended with timestamps indicating at what time a particular product had a certain price. When creating the instance of *SlowlyChangingDimension* information about how these timestamps should be interpreted is provided to the constructor. In this case, is it fairly simple as the timestamp provided in the input data is simple enough to be converted directly to `datetime.date` object. These can then be inserted into a column of type Date. To automate this conversion, the parameter `fromfinder` is set to the function returned by `pygrametl.datareader()` which constructs `datetime.date` objects from the `str` in date. However, a user-defined function with the same interface as the function generated by `pygrametl.datareader()` could also be used. When inserting the rows the method *SlowlyChangingDimension.scdensure()* is used instead of `SlowlyChangingDimension.insert()` as it first performs a lookup to verify that an existing version of the row is not already present. If a row is already present, this row is updated with the from timestamp inserted into its to time attribute indicating when this version of the row was deemed obsolete, and an incremented version number is added to the new row indicating that this is a newer version of an existing row.

2.3.7 SnowflakedDimension

SnowflakedDimension represents a snowflaked dimension as a single object with the same interface as *Dimension*. Instantiating a *SnowflakedDimension* is however different. Instead of requiring all arguments to be passed to the constructor of *SnowflakedDimension* itself, dimension objects must be created for each table in the snowflaked dimension. These objects are then passed to *SnowflakedDimension* constructor. The dimension objects must be given as pairs that indicate their foreign key relationships, e.g. (a1, a2) should be passed if a1 has a foreign key to a2. Currently, it is a requirement that a foreign key must have the same name as the primary key it references. The only additional configuration supported by *SnowflakedDimension* is `expectboguskeyvalues` which indicates that a key used as a lookup attribute in a lower level of the hierarchy might not have a matching primary key. Support for slowly changing dimensions of type 2 or a combined of type 1 and type 2 are provided if an instance of *SlowlyChangingDimension* is at the root of the snowflaked dimension. Currently, only the root dimension can be an instance of *SlowlyChangingDimension* and the feature should be considered experimental.

```
import psycopg2
import pygrametl
from pygrametl.tables import CachedDimension, SnowflakedDimension

# Input is a list of rows which in pygrametl is modeled as dicts
products = [
    {'name': 'Calvin and Hobbes 1', 'category': 'Comic',
     'type': 'Fiction', 'price': '10'},
    {'name': 'Calvin and Hobbes 2', 'category': 'Comic',
     'type': 'Fiction', 'price': '10'},
    {'name': 'Calvin and Hobbes 3', 'category': 'Comic',
     'type': 'Fiction', 'price': '10'},
    {'name': 'Cake and Me', 'category': 'Cookbook',
     'type': 'Non-Fiction', 'price': '15'},
    {'name': 'French Cooking', 'category': 'Cookbook',
     'type': 'Non-Fiction', 'price': '50'},
    {'name': 'Sushi', 'category': 'Cookbook',
     'type': 'Non-Fiction', 'price': '30'},
    {'name': 'Nineteen Eighty-Four', 'category': 'Novel',
     'type': 'Fiction', 'price': '15'},
    {'name': 'The Lord of the Rings', 'category': 'Novel',
     'type': 'Fiction', 'price': '60'}
]

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)

# The product dimension is in the database represented as a snowflaked
# dimension, so a CachedDimension object is created for each table
productTable = CachedDimension(
    name='product',
    key='productid',
    attributes=['name', 'price', 'categoryid'],
    lookupatts=['name'])

categoryTable = CachedDimension(
    name='category',
    key='categoryid',
    attributes=['category', 'typeid'],
    lookupatts=['category'])

typeTable = CachedDimension(
    name='type',
    key='typeid',
    attributes=['type'])

# An instance of SnowflakedDimension is initialized with the created
```

(continues on next page)

(continued from previous page)

```
# dimensions as input. Thus allowing a snowflaked dimension to be used in
# the same manner as a dimension consisting of a single table. The dimension's
# tables are passed as pairs based on their foreign key relationships.
# Meaning the arguments indicate that the productTable has a foreign key
# relationship with the categoryTable, and the categoryTable has a foreign
# key relationship with the typeTable. If a table has multiple foreign key
# relationships to tables in the snowflaked dimension, a list must be passed
# as the second part of the tuple with a dimension object for each table the
# first argument references through its foreign keys.
productDimension = SnowflakedDimension(references=[
    (productTable, categoryTable), (categoryTable, typeTable)])

# SnowflakedDimension provides the same interface as the dimensions classes,
# however, some of its methods stores keys in the row when they are computed
for row in products:
    productDimension.insert(row)

# Ensures that the data is committed and the connections are closed correctly
conn.commit()
conn.close()
```

In the above example, the product dimension is not represented by a single table like in the examples shown for the other dimensions classes provided by pygrametl. It is instead represented as a snowflake consisting of multiple tables. To represent this in the ETL flow, a combination of *SnowflakedDimension* and *CachedDimension* is used. As multiple tables need to be represented, an instance of *CachedDimension* is created for each. An instance of *SnowflakedDimension* is then created to represent the many instances of *CachedDimension* as a single object. Interacting with a snowflaked dimension is done through the same interface provided by the other dimension classes in pygrametl. However, some of *SnowflakedDimension* methods modify the provided rows as foreign key relationship needs to be computed based on the contents of the rows the object operates on.

```
import psycopg2
import pygrametl
from pygrametl.tables import CachedDimension, SnowflakedDimension, \
    SlowlyChangingDimension

# Input is a list of rows which in pygrametl is modeled as dicts
products = [
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '20',
     'date': '1990-10-01'},
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '10',
     'date': '1990-12-10'},
    {'name': 'Calvin and Hobbes', 'category': 'Comic', 'price': '20',
     'date': '1991-02-01'},
    {'name': 'Cake and Me', 'category': 'Cookbook', 'price': '15',
     'date': '1990-05-01'},
    {'name': 'French Cooking', 'category': 'Cookbook', 'price': '50',
     'date': '1990-05-01'},
    {'name': 'Sushi', 'category': 'Cookbook', 'price': '30',
     'date': '1990-05-01'},
    {'name': 'Nineteen Eighty-Four', 'category': 'Novel', 'price': '15',
     'date': '1990-05-01'},
    {'name': 'The Lord of the Rings', 'category': 'Novel', 'price': '60',
```

(continues on next page)

(continued from previous page)

```
     'date': '1990-05-01'}
]

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)

# The dimension is snowflaked into two tables, one with categories, and the
# other at the root with name and the slowly changing attribute price
productTable = SlowlyChangingDimension(
    name='product',
    key='productid',
    attributes=['name', 'price', 'validfrom', 'validto', 'version',
                'categoryid'],
    lookupatts=['name'],
    fromatt='validfrom',
    fromfinder=pygrametl.datereader('date'),
    toatt='validto',
    versionatt='version')

categoryTable = CachedDimension(
    name='category',
    key='categoryid',
    attributes=['category'])

productDimension = SnowflakedDimension(references=[(productTable,
                                                     categoryTable)])

# Using a SlowlyChangingDimension with a SnowflakedDimension is done in the
# same manner as a normal SlowlyChangingDimension using scdensure
for row in products:
    productDimension.scdensure(row)

# Ensures that the data is committed and the connections are closed correctly
conn.commit()
conn.close()
```

A *SlowlyChangingDimension* and a *SnowflakedDimension* can be combined if necessary, with the restriction that all slowly changing attributes must be placed in the root table. To do this, the *CachedDimension* instance connecting to the root table has to be changed to an instance of *SlowlyChangingDimension* and the necessary attributes added to the database table. Afterward, *SnowflakedDimension.scdensure()* can be used to insert and lookup rows while ensuring that the slowly changing attributes are updated correctly.

2.4 Fact Tables

pygrametl provides multiple classes for representing fact tables. These classes enable facts to be loaded one at a time, as batches stored in memory, or in bulk from a file on disk. Support for loading facts with missing information and then updating them later is also supported. For information about how to load facts in parallel see *Parallel*. In the following examples, we use PostgreSQL as the RDBMS and psycopg2 as the database driver.

All of the following classes are currently implemented in the *pygrametl.tables* module.

2.4.1 FactTable

The most basic class for representing a fact table is *FactTable*. Before creating a *FactTable* object, an appropriate table must be created in the database, and a **PEP 249** connection to the database must be created and wrapped by the class *ConnectionWrapper*. For more information about how database connections are used in pygrametl see *Database*. *FactTable* constructor must be given the table's `name`, the attributes used as `measures` in the fact table, and the attributes referencing dimensions (`keyrefs`). Be aware that *FactTable* performs an insert in the database whenever the *FactTable.insert()* method is called, which can very quickly become a bottleneck.

```
import psycopg2
import pygrametl
from pygrametl.tables import FactTable

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)

# This instance of FactTable connects to the table facttable in the
# database using the default connection wrapper created above
factTable = FactTable(
    name='facttable',
    measures=['price'],
    keyrefs=['storeid', 'productid', 'dateid'])
```

The above example shows the three step process needed to connect an instance of *FactTable* to an existing database table. Firstly, a **PEP 249** connection to the database is created. Then an instance of *ConnectionWrapper* is created to provide a uniform interface to all types of database connections supported by pygrametl. The instance of *ConnectionWrapper* is also set as the default database connection to use for this ETL flow. Lastly, a *FactTable* is created as a representation of the actual database table.

Operations on the fact table are done using three methods: *FactTable.insert()* inserts new facts directly into the fact table when they are passed to the method. *FactTable.lookup()* returns a fact if the database contains one with the given combination of keys referencing the dimensions. *FactTable.ensure()* combines *FactTable.lookup()* and *FactTable.insert()* by ensuring that a fact does not exist before inserting it. An example of each function and the automatic name mapping can be seen below, where the fact table from the last example is reused.

```
import psycopg2
import pygrametl
from pygrametl.tables import FactTable
```

(continues on next page)

(continued from previous page)

```
# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)

# This instance of FactTable connects to the table facttable in the
# database using the default connection wrapper created above
factTable = FactTable(
    name='facttable',
    measures=['price'],
    keyrefs=['storeid', 'productid', 'dateid'])

# A list of facts ready to inserted into the fact table
facts = [{'storeid': 1, 'productid': 13, 'dateid': 4, 'price': 50},
         {'storeid': 2, 'productid':  7, 'dateid': 4, 'price': 75},
         {'storeid': 1, 'productid':  7, 'dateid': 4, 'price': 50},
         {'storeid': 3, 'productid':  9, 'dateid': 4, 'price': 25}]

# The facts can be inserted using the insert method
for row in facts:
    factTable.insert(row)
conn.commit()

# Lookup returns the keys and measures given only the keys
row = factTable.lookup({'storeid': 1, 'productid': 13, 'dateid': 4})

# Ensure should be used when loading facts that might already be loaded
newFacts = [{'storeid': 2, 'itemid':  7, 'dateid': 4, 'price': 75},
            {'storeid': 1, 'itemid':  7, 'dateid': 4, 'price': 50},
            {'storeid': 1, 'itemid':  2, 'dateid': 7, 'price': 150},
            {'storeid': 3, 'itemid':  3, 'dateid': 6, 'price': 100}]

for row in newFacts:
    # The second argument forces ensure to not only match the keys for facts
    # to be considered equal, but also checks if the measures are the same
    # for facts with the same key, and if not raises a ValueError. The third
    # argument renames itemid to productid using a name mapping
    factTable.ensure(row, True, {'productid': 'itemid'})
conn.commit()
conn.close()
```

2.4.2 BatchFactTable

BatchFactTable loads facts into the fact table in batches instead of one at a time like *FactTable*. Thus reducing the number of round trips to the database which improves the performance of the ETL flow. The size of each batch is determined by the `batchsize` parameter added to the class's constructor. *BatchFactTable* loads each batch using either the `executemany()` method specified in **PEP 249** or a single SQL `INSERT INTO facttable VALUES(...)` statement depending on the value passed to `usemultirow` in the classes constructor. The *ConnectionWrapper.commit()* method must be called after all facts have been inserted into the fact table to both ensure that the last batch is loaded into the database from memory and that the transaction is committed.

Note: Both `BatchFactTable.lookup()` and `BatchFactTable.ensure()` force the current batch of facts to be an inserted. This is to keep them consistent with all of facts inserted into the fact table. Thus using these methods can reduce the benefit of batching insertions.

2.4.3 BulkFactTable

BulkFactTable also inserts facts in batches but writes the facts to a temporary file instead of keeping them in memory. Thus the size of a batch is limited by the size of the disk instead of the amount of memory available. However, this prevents `BulkFactTable.lookup()` and `BulkFactTable.ensure()` from being implemented efficiently, so these methods are not available. Like for *BatchFactTable*, the method *ConnectionWrapper.commit()* must be called to ensure that the last batch of facts is loaded into the database. Multiple additional parameters have been added to the class's constructor to provide control over the temporary file used to store facts, such as what delimiters to use and the number of facts to be bulk loaded in each batch. All of these parameters have a default value except for `bulkloader`. This parameter must be passed a function that will be called for each batch of facts to be loaded. This is necessary as the exact way to perform bulk loading differs from RDBMS to RDBMS.

`func(name, attributes, fieldsep, rowsep, nullval, filehandle):`

Required signature of a function bulk loading data from a file into an RDBMS in pygrametl. For more information about bulk loading see *Bulk Loading*.

Arguments:

- name: the name of the fact table in the data warehouse.
- attributes: a list containing both the sequence of attributes constituting the primary key of the fact table, as well as the measures.
- fieldsep: the string used to separate fields in the temporary file.
- rowsep: the string used to separate rows in the temporary file.
- nullval: if the *BulkFactTable* was passed a string to substitute None values with, then it will be passed, if not then None is passed.
- filehandle: either the name of the file or the file object itself, depending upon the value of `BulkFactTable.usefilename`. Using the filename is necessary if the bulk loading is invoked through SQL (instead of directly via a method on the PEP249 driver). It is also necessary if the bulkloader runs in another process.

In the following example, a *BulkFactTable* is used to bulk load facts into a data warehouse using function `pgbulkloader()`. For information about how to bulk loading data into other RDBMSs see *Bulk Loading*.

```
import psycopg2
import pygrametl
from pygrametl.tables import BulkFactTable
```

(continues on next page)

(continued from previous page)

```
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

conn = pygrametl.ConnectionWrapper(connection=pgconn)

facts = [{'storeid': 1, 'productid': 13, 'dateid': 4, 'price': 50},
         {'storeid': 2, 'productid':  7, 'dateid': 4, 'price': 75},
         {'storeid': 1, 'productid':  7, 'dateid': 4, 'price': 50},
         {'storeid': 3, 'productid':  9, 'dateid': 4, 'price': 25}]


# This function bulk loads a file into PostgreSQL using psycopg2
def pgbulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):
    cursor = conn.cursor()
    # psycopg2 does not accept the default value used to represent NULL
    # by BulkDimension, which is None. Here this is ignored as we have no
    # NULL values that we wish to substitute for a more descriptive value
    cursor.copy_from(file=filehandle, table=name, sep=fieldsep,
                     columns=attributes)


# The bulk loading function must be passed to BulkFactTable's constructor
factTable = BulkFactTable(
    name='facttable',
    measures=['price'],
    keyrefs=['storeid', 'productid', 'dateid'],
    bulkloader=pgbulkloader)

# commit() and close() must be called to ensure that all facts have been
# inserted into the database and that the connection is closed correctly
#  afterward
for row in facts:
    factTable.insert(row)
conn.commit()
conn.close()
```

2.4.4 AccumulatingSnapshotFactTable

AccumulatingSnapshotFactTable represents a fact table where facts are updated as a process evolves. Typically different date references (OrderDate, PaymentDate, ShipDate, DeliveryDate, etc.) are set when they become known. Measures (e.g., measuring the lag between the different dates) are also often set as they become available. Like for *FactTable*, the class *AccumulatingSnapshotFactTable* performs an insert in the database whenever the `AccumulatingSnapshotFactTable.insert()` method is called. The following example illustrates how to create the class:

```
import psycopg2
import pygrametl
from pygrametl.tables import AccumulatingSnapshotFactTable

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
```

(continues on next page)

(continued from previous page)

```
                          password='dwpass'""")

# This ConnectionWrapper will be set as a default and is then implicitly
# used, but it is stored in conn so transactions can be committed and the
# connection closed
conn = pygrametl.ConnectionWrapper(connection=pgconn)


# A factexpander can be used to modify a row only if it has been updated, note
# that we only ignore namemapping for brevity, production code should use it
def computelag(row, namemapping, updated):
    if 'shipmentdateid' in updated:
        row['shipmentlag'] = row['shipmentdateid'] - row['paymentdateid']
    if 'deliverydateid' in updated:
        row['deliverylag'] = row['deliverydate'] - row['shipmentdateid']


# This instance of AccumulatingSnapshotFactTable connects to the table
# orderprocessing in the database using the connection created above
asft = AccumulatingSnapshotFactTable(
    name='orderprocessing',
    keyrefs=['orderid', 'customerid', 'productid'],
    otherrefs=['paymentdateid', 'shipmentdateid', 'deliverydateid'],
    measures=['price', 'shipmentlag', 'deliverylag'],
    factexpander=computelag)
```

Firstly a **PEP 249** connection is created to perform the actual database operations, then an instance of the *ConnectionWrapper* is created as a uniform wrapper around the **PEP 249** connection which is set as the default database connection for this ETL flow. Then a user-defined function to compute lag measures is defined. Lastly, an *AccumulatingSnapshotFactTable* is created.

As stated `AccumulatingSnapshotFactTable.insert()` inserts new facts directly into the fact table when they are passed to the method. `AccumulatingSnapshotFactTable.lookup()` checks if the database contains a fact with the given combination of keys referencing the dimensions. These methods behave in the same way as in *FactTable*. The method *AccumulatingSnapshotFactTable.update()*, will based on the `keyrefs`, find the fact and update it if there are any differences in `otherrefs` and `measures`. The method *AccumulatingSnapshotFactTable.ensure()* checks if the row it is given, already exists in the database table. If it does not exist, it is immediately inserted. If it exists, the method will see if some of the values for `otherrefs` or `measures` have been updated in the passed row. If so, it will update the row in the database. Before that, it will, however, run the `factexpander()` if one was given to `AccumulatingSnapshotFactTable.__init__()` when the object was created. Note that the generated SQL for lookups and updates will use the `keyrefs` in the `WHERE` clause and an index on them should be considered. An example of how to use the class can be seen below:

```
import psycopg2
import pygrametl
from pygrametl.tables import AccumulatingSnapshotFactTable

# The actual database connection is handled by a PEP 249 connection
pgconn = psycopg2.connect("""host='localhost' dbname='dw' user='dwuser'
                          password='dwpass'""")

# A factexpander can be used to modify a row only if it has been updated, note
# that we only ignore namemapping for brevity, production code should use it
```

(continues on next page)

(continued from previous page)

```
conn = pygrametl.ConnectionWrapper(connection=pgconn)


# A factexpander can be used to modify a row only if it has been updated, note
# that we only ignore namemapping for brevity, production code should use it
def computelag(row, namemapping, updated):
    if 'shipmentdateid' in updated:
        row['shipmentlag'] = row['shipmentdateid'] - row['paymentdateid']
    if 'deliverydateid' in updated:
        row['deliverylag'] = row['deliverydate'] - row['shipmentdateid']


# This instance of AccumulatingSnapshotFactTable connects to the table
# orderprocessing in the database using the connection created above
asft = AccumulatingSnapshotFactTable(
    name='orderprocessing',
    keyrefs=['orderid', 'customerid', 'productid'],
    otherrefs=['paymentdateid', 'shipmentdateid', 'deliverydateid'],
    measures=['price', 'shipmentlag', 'deliverylag'],
    factexpander=computelag)

# A list of facts that are ready to inserted into the fact table
facts = [{'orderid': 1, 'customerid': 1, 'productid': 1, 'price': 10},
         {'orderid': 2, 'customerid': 2, 'productid': 2, 'price': 20},
         {'orderid': 3, 'customerid': 3, 'productid': 3, 'price': 30}]

# The facts can be inserted using the ensure method. (If we had used the
# insert method instead, we should have made sure the facts above had a
# value for each attribute in the fact table. When using ensure, missing
# attributes will be set to None before an insertion.)
for row in facts:
    asft.ensure(row)

# Now assume that the the orders get paid and shipped
facts[0]['paymentdateid'] = 12
facts[0]['shipmentdateid'] = 14
facts[2]['paymentdateid'] = 11

# Update the accumulating fact table in the DW
for row in facts:
    asft.ensure(row)  # will call computelag and do the needed updates

conn.commit()
conn.close()
```

2.5 Bulk Loading

Bulk loading rows instead of inserting them one at a time can dramatically increase the throughput of an ETL program. Bulk loading works by loading data from a temporary file into the database. The actual process of bulk loading is unfortunately different for each RDBMS. Because of this, a user-defined function must be created that uses the functionality provided by a particular RDBMS to bulk load the data from a file. The following is a list of example functions showing how bulk loading can be performed for some of the more commonly used RDBMSs.

Currently, three classes in pygrametl use bulk loading: *BulkDimension*, *CachedBulkDimension*, and *BulkFactTable*. Thus a function that can bulk load data from a file into the specific RDBMS used for the data warehouse, must be passed to each of these classes constructors. The function must have the following signature:

func(name, attributes, fieldsep, rowsep, nullval, filehandle):

Required signature of a function bulk loading data from a file into an RDBMS in pygrametl.

Parameters

- **name** – The name of the table in the data warehouse.
- **attributes** – A list containing the sequence of attributes in the table.
- **fieldsep** – The string used to separate fields in the temporary file.
- **rowsep** – The string used to separate rows in the temporary file.
- **nullval** – If the class was passed a string to substitute None values with, then it will be passed, if not then None is passed.
- **filehandle** – Either the name of the file or the file object itself, depending upon the value of member `usefilename` on the class.

2.5.1 PostgreSQL

For PostgreSQL the copy_from method from psycopg2 can be used:

```
# psycopg2
def pgbulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):
    global connection
    cursor = connection.cursor()
    cursor.copy_from(file=filehandle, table=name, sep=fieldsep, null=nullval,
                         columns=attributes)
```

If Jython is used the copyIn method in JDBC's `CopyManager` class can be used:

```
# JDBC
def pgcopybulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):
    global pgconnection
    copymgr = pgconnection.getCopyAPI()
    sql = "COPY %s(%s) FROM STDIN WITH DELIMITER '%s'" % \
          (name, ', '.join(attributes), fieldsep)
    copymgr.copyIn(sql, filehandle)
```

2.5.2 MySQL

For MySQL the LOAD DATA INFILE functionality provided by MySQL SQL dialect can be used.

```
# MySQLdb
def mysqlbulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):
    global connection
    cursor = connection.cursor()
    sql = "LOAD DATA LOCAL INFILE '%s' INTO TABLE %s FIELDS TERMINATED BY '%s' LINES␣
↪TERMINATED BY '%s' (%s);" % \
            (filehandle, name, fieldsep, rowsep, ', '.join(attributes))
    cursor.execute(sql)
```

2.5.3 Oracle

Oracle supports two methods for bulk loading from text files, SQL Loader and External Tables. The following example uses SQL Loader as it does not require the creation of an additional table, which is problematic to do in a bulk loading function as the data types of each column must be specified.

SQL Loader is part of Oracle's client package. SQL Loader requires all configuration and data files to have specific suffixes, so a file must be created with the suffix .dat and passed to any bulk loading table as `tempdest`.

```
with tempfile.NamedTemporaryFile(suffix=".dat") as dat_handle:
    BulkDimension(
        ...
        tempdest=dat_handle)
```

The bulk loading function shown below constructs a control file with the .ctl suffix using the functions arguments. The SQL Loader is then executed (sqlldr must in the system path) and passed the constructed .ctl file.

```
# cx_Oracle or JDBC
def oraclebulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):

    # The configuration file used by SQL Loader must use the suffix .ctf
    with tempfile.NamedTemporaryFile(suffix=".ctl") as ctl_handle:

        # The attributes to be loaded must be qouted using double quotes
        unqouted_atts = str(tuple(attributes)).replace("'", "")
        ctl_contents = """
            LOAD DATA INFILE '%s' "str %r"
            APPEND INTO TABLE %s
            FIELDS TERMINATED BY %r
            %s
            """ % (filehandle.name, rowsep, name, fieldsep, unqouted_atts)

        # Strips the multi line string of unnecessary indention, and ensures
        # that the contents are written to the file by flushing it
        ctl_contents = textwrap.dedent(ctl_handle).lstrip()
        ctl_handle.write(ctl_contents)
        ctl_handle.flush()

        # Bulk loads the data using Oracle's SQL Loader. As a new connection
        # is created, the same username, passowrd, etc. must be given again
```

(continues on next page)

(continued from previous page)

```
        os.system("sqlldr username/password@ip:port/sid control=" +
                  str(ctl_handle.name))
```

2.5.4 Microsoft SQL Server

For Microsoft SQL Server the BULK INSERT functionality provided by Transact-SQL can be used.

There are a number of things to be aware of when using pygrametl with SQL Server. If the file used for bulk loading is located on a machine running Microsoft Windows, the file must be copied before bulk loading, as the locks placed on the file by the OS and pygrametl, prevent SQL Server from opening it directly. Copying the file can be done e.g. using shutil.copyfile.

By default, BULK INSERT ignores column names, so the number and order of columns must match the table you are inserting into. This can be overcome by adding a format file. In this case, we create a non-XML format file.

A simple example of bulk loading in SQL Server along with the creation of a format file can be seen below:

```
def sqlserverbulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):
    global msconn
    cursor = msconn.cursor()

    # Copy the tempdest
    shutil.copyfile(filehandle, r'd:\dw\tmpfilecopy')

    # Create format file
    fmt = open(r'd:\dw\format.fmt', 'w+')
    # 12.0 corresponds to the version of the bcp utility being used by SQL Server.
    # For more information, see the above link on non-XML format files.
    fmt.write("12.0\r\n%d\r\n" % len(attributes))
    count = 0
    sep = "\\t"
    for a in attributes:
        count += 1
        if count == len(attributes): sep = "\\n"
        # For information regarding the format values,
        # see the above link on non-XML format files.
        fmt.write('%d SQLCHAR 0 8000 "%s" %d %s "Latin1_General_100_CI_AS_SC"\r\n' %␣
↪(count, sep, count, a))
    fmt.close()

    sql = "BULK INSERT %s FROM '%s' WITH (FORMATFILE = '%s', FIELDTERMINATOR = '%s',␣
↪ROWTERMINATOR = '%s')" % \
            (name, r'd:\dw\tmpfilecopy', r'd:\dw\format.fmt', fieldsep, rowsep,)
    cursor.execute(sql)
```

2.6 Parallel

pygrametl provides multiple abstractions to simplify the creation of parallel ETL flow, to take advantage of modern multi-core and multi-processor systems. Firstly, any *datasources* can be read in a separate process using *ProcessSource*. Further parallelism can be archived by decoupling tables from the main process and allowing these decoupled tables to communicate with each other without interrupting the main process. Tables can also be partitioned so operations on large tables can be performed by multiple processes. Both decoupled tables and partitioning tables can be found in the *tables* module. To support database connections from multiple decoupled tables any *ConnectionWrapper* and *JDBCConnectionWrapper* must be wrapped by the function *shareconnectionwrapper()* before being used by multiple decoupled tables.

pygrametl also provides abstractions for running functions in parallel. The decorator *splitpoint()* can be used to annotate functions that should run in separate processes. This supplements the decoupled tables, as many transformations are done in a set of functions before they are inserted into a database table. Splitpoints can be synchronized using the function *endsplits()*. The function *createflow()* can be used to create a sequence of functions that run in separate processes. In a flow a row is first given to the first function, then the second, and so forth. This also means the passed row must be modified as the functions return values are ignored.

Note: pygrametl supports executing parallel ETL flows using CPython (only on platforms that start new processes using fork) or Jython. The method used by CPython to start a process can be determined using `multiprocessing.get_start_method()`. Unix-like operating systems generally use *fork* by default, but some must be configured to use *fork* through `multiprocessing.set_start_method()`. Microsoft Windows does not support *fork*, thus a Unix-like environment must be installed, e.g., using Windows Subsystem for Linux.

Due to CPython's GIL, Jython should be used to run ETL flows that use pygrametl parallel constructs. This is because Jython allows threads to be used for parallel processing, while it is necessary to use processes in CPython. Thus the term process is used to denote a process or a thread, depending on the Python implementation in question. For more information on using pygrametl on Jython, see *Jython*.

2.6.1 ProcessSource

ProcessSource is a data source that allows other data sources to be iterated through in a separate process. A data source in pygrametl is a set of abstraction that provides access to multiple types of data through a normal Python iterator. For more information about data sources see *Data Sources*.

```
from pygrametl.tables import FactTable, CachedDimension
from pygrametl.datasources import CSVSource, ProcessSource, \
        TransformingSource
from pygrametl.JDBCConnectionWrapper import JDBCConnectionWrapper

# JDBC and Jython are used as threads usually provide better performance
import java.sql.DriverManager
jconn = java.sql.DriverManager.getConnection(
    "jdbc:postgresql://localhost/dw?user=dwuser&password=dwpass")
conn = JDBCConnectionWrapper(jdbcconn=jconn)

factTable = FactTable(
    name='facttable',
    measures=['sale'],
    keyrefs=['storeid', 'productid', 'dateid'])
```

(continues on next page)

(continued from previous page)

```
productTable = CachedDimension(
        name='product',
        key='productid',
        attributes=['name', 'price'],
        lookupatts=['name'])


# A set of computational expensive functions are needed to transform the
# facts before they can be inserted into the fact table. Each function must
# be defined as func(row) so a TransformationSource can combine them before
# they are passed to ProcessSource and run in another thread
def convertReals(row):
    # Converting a string encoding of a float to an integer must be done in
    # two steps, first it must be converted to a float and then to an integer
    row['sale'] = int(float(row['sale']))


def trimProductname(row):
    row['name'] = row['name'].strip()


# In the transformation we use three data sources to retrieve rows from
# sales.csv, first CSVSource to read the csv file, then
# TransformationSource to transform the rows, and lastly ProcessSource to
# do both the reading and transformation in another thread
sales = CSVSource(f=open('sales.csv'), delimiter=',')
transSales = TransformingSource(sales, convertReals, trimProductname)
salesProcess = ProcessSource(transSales)

# While the list of sales are being read and transformed by the spawned
# thread, the main thread is occupied with pre-loading the product dimension
# with data from product.csv
products = CSVSource(f=open('product.csv'), delimiter=',')
for row in products:
    productTable.insert(row)

# After the ProcessSource have read rows from the data source provided, they
# can be accessed through ProcessSource iterator like any other data source
for row in salesProcess:
    row['productid'] = productTable.lookup(row)
    factTable.insert(row)
conn.commit()
conn.close()
```

In the above example, we use a *ProcessSource* to transform a set of rows from sales.csv while we fill the product dimension with data. As the use of a *ProcessSource* adds additional overhead to the iterator, seeing as rows must be transferred in batches from another process, other computations should be performed in between the creation and use of the data source to allow for data to be read, transformed, and transferred.

2.6.2 Decoupled Tables

A decoupled table in pygrametl is a proxy for an instance of another table class defined in the *tables* module. Currently, two different classes exist for decoupled tables, *DecoupledDimension* and *DecoupledFactTable*. The two classes behave nearly identically with one implementing the interface of a dimension and the other the interface of a fact table. When a method is called on one of the two classes, a message is sent to the actual table object, and if the method has a return value an instance of the class `FutureResult` is returned. This instance is a handle to the actual result when it becomes available. To get the actual result, the instance can be given directly to a method accepting a row which would force the method to block until a value is ready, or the entire decoupled can be consumed by another decoupled table. When a decoupled table is consumed by another decoupled table, the values are extracted from an instance of `FutureResult` by the table that needs it without blocking the caller of methods on that table. It should however be noted that any rows passed to an instance of *DecoupledFactTable* or *DecoupledDimension* should only contain the attributes directly needed by the table, as having additional key/value pairs in the `dict` can make pygrametl insert the row before the actual values are ready, leading to instances of the class `FutureResult` being incorrectly passed to the database instead.

```
from pygrametl.datasources import CSVSource
from pygrametl.tables import FactTable, CachedDimension,\
     DecoupledDimension, DecoupledFactTable
from pygrametl.JDBCConnectionWrapper import JDBCConnectionWrapper
from pygrametl.parallel import shareconnectionwrapper

# The data is read from a csv file
inputdata = CSVSource(f=open('sales.csv', 'r'), delimiter=',')

# JDBC and Jython are used as threads usually provide better performance
import java.sql.DriverManager
jconn = java.sql.DriverManager.getConnection(
    "jdbc:postgresql://localhost/dw?user=dwuser&password=dwpass")

# The connection wrapper is itself wrapped in a SharedConnectionClient,
# so it can be shared by multiple decoupled tables in a safe manner
conn = JDBCConnectionWrapper(jdbcconn=jconn)
shrdconn = shareconnectionwrapper(targetconnection=conn)

# The product dimension is decoupled and runs in a separate thread allowing
# it to be accessed by other decoupled tables without using the main thread
productDimension = DecoupledDimension(
    CachedDimension(
        name='product',
        key='productid',
        attributes=['name', 'price'],
        lookupatts=['name'],
        # The SharedConnectionWrapperClient must be copied for each
        # decoupled table that use it correct interaction with the database
        targetconnection=shrdconn.copy(),
        prefill=True)
    )

# The fact table is also decoupled in order to consume the values returned
# from the methods called on the product dimension without blocking the main
# thread while waiting for the database. Thus allowing the main thread to
# perform other operations needed before a full fact is ready
```

(continues on next page)

(continued from previous page)

```
factTable = DecoupledFactTable(
    FactTable(
        name='facttable',
        measures=['sale'],
        keyrefs=['storeid', 'productid', 'dateid'],
        targetconnection=shrdconn.copy()),
    returnvalues=False,
    consumes=[productDimension]
    )

# Inserting facts into the database can be done in the same manner as in a
# sequential ETL flow, extraction of data from the product dimension is
# done automatically by pygrametl
for row in inputdata:
    # A new row is created for each fact, as having values not present in a
    # decoupled table that consumes another dimension, can make pygrametl
    # miscalculate when the actual results are ready, making the framework
    # pass a FutureResult to the database which usually raises an error
    fact = {}
    fact['storeid'] = row['storeid']
    fact['productid'] = productDimension.ensure(row)
    fact['dateid'] = row['dateid']
    fact['sale'] = row['sale']
    # Other CPU intensive transformations should be performed to take
    # advantage of the decoupled dimensions automatically exchanging data
    factTable.insert(fact)
shrdconn.commit()
shrdconn.close()
```

The above example shows a very simple use of decoupled tables in pygrametl, for real-world application, tuning of queues and buffers should be done to match the underlying hardware to maximize the performance of the parallel ETL flow. Although the example uses an instance of *Dimension* and *FactTable* for simplicity, it is supported for all types of dimensions and fact tables, except *SubprocessFactTable* on CPython as it already runs in its own process. Decoupling of tables requiring a large amount of processing when their methods are called, like a *SnowflakedDimension*, can help increase performance due to not blocking the main process while waiting on the database performing the joins.

If any user-defined function needs to access the database and be synchronized with the decoupled tables, it must be passed to *shareconnectionwrapper()*. An example of such a function is the bulk loader used for pygrametl's *BulkFactTable*.

```
from pygrametl.JDBCConnectionWrapper import JDBCConnectionWrapper
from pygrametl.parallel import shareconnectionwrapper

# JDBC and Jython is used as threads usually provides better performance
import java.sql.DriverManager
jconn = java.sql.DriverManager.getConnection(
    "jdbc:postgresql://localhost/dw?user=dwuser&password=dwpass")


# A user-defined function that can bulk load data into PostgreSQL over JDBC
def bulkloader(name, attributes, fieldsep, rowsep, nullval, filehandle):
    global jconn
    copymgr = jconn.getCopyAPI()
```

(continues on next page)

(continued from previous page)

```
    sql = "COPY %s(%s) FROM STDIN WITH DELIMITER '%s'" % \
          (name, ', '.join(attributes), fieldsep)
    copymgr.copyIn(sql, filehandle)


# The connection wrapper is itself wrapped in a SharedConnectionClient so it
# can be shared by multiple decoupled tables in a safe manner. The function
# bulkloader is given to shareconnectionwrapper so the shared connection
# wrapper can ensure that the bulk loading function is synchronized with
# the decoupled tables using the shared connection wrapper
conn = JDBCConnectionWrapper(jdbcconn=jconn)
scw = shareconnectionwrapper(targetconnection=conn, userfuncs=[bulkloader])
```

2.6.3 Partitioning Tables

If a particular dimension or fact table requires more processing than the other tables, it can be beneficial to partition it into multiple partitions. Thus allowing operations to be conducted on one table in parallel to reduce the time needed to process that particular table. pygrametl supports partitioning of tables through multiple features. Firstly, the classes *DimensionPartitioner* and *FactTablePartitioner* automates the partitioning of rows for multiple decoupled dimensions or fact tables. How to do the partitioning is determined by a partitioning function with the signature `func(dict)()`. If no function is passed, then a default partitioning function is used as documented in the API. Secondly, to ensure that unique surrogate keys are assigned to all rows in a partitioned table, a shared sequence factory can be created using the function *getsharedsequencefactory()*. Each parallel process is then given a unique set of numbers to use as surrogate keys, ensuring that all surrogate keys are unique despite being assigned by separate processes.

```
from pygrametl.datasources import CSVSource
from pygrametl.tables import FactTable, CachedDimension, \
    DecoupledDimension, DecoupledFactTable, DimensionPartitioner
from pygrametl.parallel import shareconnectionwrapper, \
    getsharedsequencefactory
from pygrametl.JDBCConnectionWrapper import JDBCConnectionWrapper

sales = CSVSource(f=open('sales.csv', 'r'), delimiter=',')

# JDBC and Jython are used as threads usually provide better performance
import java.sql.DriverManager
jconn = java.sql.DriverManager.getConnection(
    "jdbc:postgresql://localhost/dw?user=dwuser&password=dwpass")

# The connection wrapper is itself wrapped in a SharedConnectionClient,
# so it can be shared by multiple decoupled tables in a safe manner
conn = JDBCConnectionWrapper(jdbcconn=jconn)
shrdconn = shareconnectionwrapper(targetconnection=conn)

# A sharedsequencefactory is created which provides values starting at zero.
# It gives each table a sequence of numbers to use as surrogate keys. The
# size of the sequence can be increased through a second argument if the
# sharedsequencefactory becomes a bottleneck in the ETL flow
idfactory = getsharedsequencefactory(0)
```

(continues on next page)

(continued from previous page)

```
# The product dimension must use the sharedsequencefactory to ensure that
# the two processes do not assign overlapping surrogate key to the rows
productDimensionOne = DecoupledDimension(
    CachedDimension(
        name='product',
        key='productid',
        attributes=['name', 'price'],
        lookupatts=['name'],
        idfinder=idfactory(),
        targetconnection=shrdconn.copy(),
        prefill=True)
    )

productDimensionTwo = DecoupledDimension(
    CachedDimension(
        name='product',
        key='productid',
        attributes=['name', 'price'],
        lookupatts=['name'],
        idfinder=idfactory(),
        targetconnection=shrdconn.copy(),
        prefill=True)
    )

# The partitioning of data is automated by the DimensionPartitioner using
# a hash on the name of product. A FactTablePartitioner is also provided
productDimension = DimensionPartitioner(
    parts=[productDimensionOne, productDimensionTwo],
    partitioner=lambda row: hash(row['name']))

# Only partitioned tables needs to use the sharedsequencefactory, normal tables
# can without any problems use the default self-incrementing surrogate key
factTable = DecoupledFactTable(
        FactTable(
            name='facttable',
            measures=['sale'],
            keyrefs=['storeid', 'productid', 'dateid'],
            targetconnection=shrdconn.copy()),
        returnvalues=False,
        # When consuming a partitioned dimension each part should be
        # consumed separately, a simple way to do so is using the parts
        # method which returns all parts managed by the partitioner
        consumes=productDimension.parts
        )

# A partitioned table can be used in the same way as any other pygrametl
# table since the framework takes care of the partitioning behind the scenes
for row in sales:
    # A new row is created for each fact, as having values not present in a
    # decoupled table that consumes another dimension, can make pygrametl
    # miscalculate when the actual results are ready, making the framework
    # pass a FutureResult to the database which usually raises an error
```

(continues on next page)

(continued from previous page)

```
    fact = {}
    fact['storeid'] = row['storeid']
    fact['dateid'] = row['dateid']
    fact['productid'] = productDimension.ensure(row)
    fact['sale'] = row['sale']
    # Other CPU intensive transformations should be performed to take
    # advantage of the decoupled dimensions automatically exchanging data
    factTable.insert(fact)
shrdconn.commit()
shrdconn.close()
```

The above example shows how to partition the data of the product dimension to multiple decoupled tables. This allows operations on the dimension to be performed by two different processes. The rows are partitioned using hash partitioning on the attribute `name`. A shared sequence factory is used to provide surrogate keys for the product dimension, as using a self-incrementing key would assign the same value to multiple rows. This is not needed for the fact table as only one table handles all operations on the fact table in the database, so a simple self-incrementing key is fine.

2.6.4 Splitpoints

As CPU intensive operations are often performed in user-defined functions, the decorator *splitpoint()* is provided. This decorator functions in much the same way as decoupled classes do for tables, as a number of processes are spawned to run the function. The number of processes to spawn can be passed to the decorator, allowing more processes to be created for functions with a longer run time. The first time a function with a decorator is called, a process is created to handle the call. This is done until the number of created processes matches the argument given to the decorator. Then, if a process is not available, the call and its arguments are added to a `queue` shared by the process created for the splitpoint. If a split function calls another function that requires synchronization it can be annotated with a new splitpoint with one as the argument, specifying that only one process is allowed to call this function at a time. To ensure all annotated functions are finished, the function *endsplits()* must be called, which joins all processes created by split points up to that point.

```
from pygrametl.tables import FactTable
from pygrametl.datasources import CSVSource
from pygrametl.parallel import splitpoint, endsplits
from pygrametl.JDBCConnectionWrapper import JDBCConnectionWrapper

sales = CSVSource(f=open('sales.csv', 'r'), delimiter=',')

# JDBC and Jython are used as threads usually provide better performance
import java.sql.DriverManager
jconn = java.sql.DriverManager.getConnection(
    "jdbc:postgresql://localhost/dw?user=dwuser&password=dwpass")

conn = JDBCConnectionWrapper(jdbcconn=jconn)

factTable = FactTable(
    name='facttable',
    measures=['sale'],
    keyrefs=['storeid', 'productid', 'dateid']
    )
```

(continues on next page)

(continued from previous page)

```
# Five threads are created to run this function, so five rows can be
# transformed at the same time. If no threads are available, the row
# is added to a queue and transformed when a thread becomes idle
@splitpoint(instances=5)
def performExpensiveTransformations(row):
    # Do some (expensive) transformations...

    # As multiple threads perform the operation inside this function. a second
    # function must be created to synchronize inserting rows into the database
    insertRowIntoData(row)


# The function is annotated with an argument-free splitpoint, so its argument
# becomes one, thereby specifying that this function should run in one thread
@splitpoint
def insertRowIntoData(row):
    factTable.insert(row)


# The CSV file is read by the main thread, then each row is transformed by
# one of five threads, before being added to the database by a sixth thread
for row in sales:
    performExpensiveTransformations(row)

# To ensure that all splitpoint annotated functions are finished before
# the ETL flow is terminated, the function endsplits must be called as it
# joins all the threads created by splitpoints up to this point
endsplits()
conn.commit()
conn.close()
```

The above example shows how to use splitpoints. Here, a very computationally expensive function is annotated with a `splitpoint` which is given the argument five, allowing five processes to run the function at the same time. The second `splitpoint` without an argument ensures that only one process is allowed to execute that function at a time, so even though it is called from `performExpensiveTransformation()` only one process can insert rows into the fact table at the same time. Should the operations on the fact table become a bottleneck, it could be partitioned using *FactTablePartitioner*. To ensure that all splitpoints have finished execution, the function *endsplits()* is executed, which joins all splitpoints, before the database connection is closed.

As splitpoint annotated functions run in separate processes, any values they return are not available to the process calling them. To work around this restriction a queue can be passed as an argument to `splitpoint` in which the split function's returned values will be added.

```
from pygrametl.datasources import CSVSource
from pygrametl.parallel import splitpoint, endsplits
from pygrametl.jythonmultiprocessing import Queue

queue = Queue()
sales = CSVSource(f=open('sales.csv', 'r'), delimiter=',')


# A queue is passed to the decorator, which uses it to store return values
@splitpoint(instances=5, output=queue)
```

(continues on next page)

(continued from previous page)

```
def expensiveReturningOperation(row):

    # Some special value, in this case None, is used to indicate that no
    # more data will be given to the queue and that processing can continue
    if row is None:
        return None

    # Returned values are automatically added to the queue for other to use
    return row


# Each row in the sales.csv is extracted and passed to the function
for row in sales:
    expensiveReturningOperation(row)

# A simple sentinel value can be used to indicate that all rows have been
# processed and that the loop using the results below can break
expensiveReturningOperation(None)

# A infinite loop is used to process the returned values as the number of
# returned rows are unknown, so a sentinel value and a break is used instead
while True:
    # Extracts the processed row returned by the annotated function, a
    # simple sentinel value is used to indicate when the processing is done
    elem = queue.get()
    if elem is None:
        break

    # Use the returned elements after the sentinel check to prevent errors
    # ......

# To ensure that all splitpoint annotated functions are finished before
# the ETL flow is terminated, the function endsplits must be called as it
# joins all the process created by splitpoints up to this point
endsplits()
```

2.6.5 Flows

Another way to parallelize transformations is to use flows. In pygrametl, a flow is a sequence of functions with the same interface, each running in its own separate process, and where each function calls the next function in the sequence. A flow can be created from multiple different functions using the *createflow()* function. After a flow is created it can be called just like any other function. Internally, the arguments are passed from the first function to the last. While the arguments are passed to the functions, any returned values are ignored. Unlike *splitpoint()*, arguments are passed in batches and not as single values to reduce the overhead of synchronization.

```
from pygrametl.tables import Dimension
from pygrametl.datasources import CSVSource
from pygrametl.parallel import splitpoint, endsplits, createflow
from pygrametl.JDBCConnectionWrapper import JDBCConnectionWrapper

# JDBC and Jython are used as threads usually provide better performance
```

(continues on next page)

(continued from previous page)

```
import java.sql.DriverManager
jconn = java.sql.DriverManager.getConnection(
    "jdbc:postgresql://localhost/dw?user=dwuser&password=dwpass")

conn = JDBCConnectionWrapper(jdbcconn=jconn)

products = CSVSource(f=open('product.csv', 'r'), delimiter=',')

productDimension = Dimension(
        name='product',
        key='productid',
        attributes=['name', 'price'],
        lookupatts=['name'])


# Two functions are defined to transform the information in product.csv
def normaliseProductNames(row):
    # Expensive operations should be performed in a flow, this example is
    # simple, so the performance gain is negated by the synchronization
    row['name'].lower()


def convertPriceToThousands(row):
    # Expensive operations should be performed in a flow, this example is
    # simple, so the performance gain is negated by the synchronization
    row['price'] = int(row['price']) / 1000


# A flow is created from the two functions defined above, this flow can then
# be called just like any other functions despite being parallelized
flow = createflow(normaliseProductNames, convertPriceToThousands)


# The data is read from product.csv in a splitpoint so the main process
# does not have to both read the input data and load it into the table
@splitpoint
def producer():
    for row in products:
        flow(row)

    # The flow should be closed when there is no more data available,
    # this means no more data is accepted but the computations will finish
    flow.close()


# The producer is called and the separate process starts to read the input
producer()

# The simplest way to extract rows from a flow is just to iterate over it,
# however additional functions to get the results as a list are available
for row in flow:
    productDimension.insert(row)
```

(continues on next page)

(continued from previous page)

```
endsplits()
conn.commit()
```

A flow is used in the above example to combine multiple functions, each transforming the rows from product.csv. By creating a flow with these functions, a process is created for each to increase the ETL flows throughput. The overhead of transferring data between the functions is reduced through batching. Rows are provided to the flow in function `producer()`, which runs in a separate process using a splitpoint so the main process can load the transformed rows into the database by iterating over the flow.

2.7 Jython

pygrametl supports running ETL flows on Jython, an implementation of Python that run on the JVM. Using Jython instead of CPython allows an ETL flow to be parallelized using multiple threads instead of multiple processes. This is because Jython does not have a global interpreter lock, which in CPython ensures that only a single thread is running per process at a given time. For more information about the GIL see the Python wiki GIL.

To make switching between CPython and Jython as simple as possible, two abstractions are provided by pygrametl. Firstly, *`JDBCConnectionWrapper`* provides two connection wrappers for JDBC connections with the same interface as the connection wrappers for **PEP 249** connections. As the connection wrappers, all share the same interface the user usually only has to change the connection type (JDBC or **PEP 249**) and the connection wrapper when switching between CPython and Jython. For more information about database access in pygrametl see *Database*. Secondly, Jython currently has no support for `multiprocessing` as threads are more lightweight than processes and multiple threads can be run in parallel. So pygrametl includes the module *`jythonmultiprocessing`* which wraps Python's `threading` module and provides a very small part of Python's `multiprocessing` module. Thus, pygrametl exposes the same interface for creating parallel ETL flows no matter if a user is using CPython or Jython.

While both Jython and CPython are capable of executing the same language, the two platforms are implemented differently, so optimizations suitable for one platform may be less effective on the other. One aspect to be aware of when running high-performance pygrametl-based ETL flows on Jython is memory management. For example, Oracle's HotSpot JVM implements a generational garbage collector that uses a much slower garbage collection strategy for the old generations than for the young. Thus, allowing too many objects to be promoted to the old generations can reduce the throughput of an ETL flow significantly. Unfortunately, this can easily occur if the values controlling caching, such as `Decoupled.batchsize`, are set too high. Similarly, if the value for `Decoupled.batchsize` is set too low the overhead of transferring data between threads increases as smaller batches are used. Many tools for profiling programs running on the JVM exist: JFR and JConsole are bundled with the JDK, while tools such as VisualVM must be installed separately but often provide additional functionality.

2.7.1 Setup

Using pygrametl with Jython requires an extra step compared to CPython, as Jython is less integrated with Python's package management system. Firstly, install pygrametl from PyPI or by downloading the development version from GitHub. For more information about installing pygrametl for use with CPython see *Install Guide*.

After pygrametl has been installed, the location it has been installed to must be added to the environment variable `JYTHONPATH`, as Jython purposely does not import modules from CPython by default. The default directory used by CPython for packages depends on the operating system and whether a package is installed locally or globally. Check the output of the `pip install` command or its log for precise information about where the package has being installed. The method for setting this variable depends on the operating system. On most Unix-like systems, the variable can be set in `~/.profile`, which will be sourced on login. On Windows, environment variables can be changed through the System setting in the Control Panel. Python's module search path can also be extended on a per program basis by adding a path to `sys.path` at the start of a Python program.

2.7.2 Usage

Jython can in most cases be used as a direct replacement for CPython unless its C API is being used. While Jython does not implement CPython C API, it can use libraries implemented in other JVM-based languages like Java, Scala, Clojure, and Kotlin. To use such libraries, they must be added to the JVM classpath by using the `-J-cp` command-line option. For more information about Jython's command-line flags run the command `jython -h`.

```
from pygrametl.tables import FactTable
from pygrametl.JDBCConnectionWrapper import JDBCConnectionWrapper

# The Java classes used must be explicitly imported into the program
import java.sql.DriverManager

# The actual database connection is handled by a JDBC connection
jconn = java.sql.DriverManager.getConnection(
    "jdbc:postgresql://localhost/dw?user=dwuser&password=dwpass")

# As PEP 249 and JDBC connections provide different interfaces, is it
# necessary to use a JDBCConnectionWrapper instead of a ConnectionWrapper.
# Both provides the same interface, thus pygrametl can execute queries
# without taking into account how the connection is implemented
conn = JDBCConnectionWrapper(jdbcconn=jconn)

# This instance of FactTable manages the table "facttable" in the
# database using the default connection wrapper created above
factTable = FactTable(
    name='testresults',
    measures=['errors'],
    keyrefs=['pageid', 'testid', 'dateid'])
```

The above example demonstrates how few changes are needed to change the first example from *Fact Tables* from using CPython to Jython. The database connection is changed from a **PEP 249** connection to a JDBC connection, and *ConnectionWrapper* is changed to *JDBCConnectionWrapper.JDBCConnectionWrapper*. The creation of the *FactTable* object does not need to be changed to run on Jython, as the connection wrappers abstract away the differences between JDBC and **PEP 249**. The other Jython module, *jythonmultiprocessing*, is even simpler to use as pygrametl's parallel module *parallel* imports either it or CPython's built-in `multiprocessing` module depending on whether Jython or CPython is used.

2.8 Drawn Table Testing

pygrametl provides the Drawn Table abstraction to simplify testing. A Drawn Table is a string-based representation of a database table. It is implemented in the Drawn Table Testing (DTT) module, but this *does not* mean that the user necessarily must implement the ETL flow itself with pygrametl or in Python – the ETL flow can be implemented using any programming language or program, including GUI-based ETL tools. First, the functionality provided by DTT is described, then how DTT can be used as a Python package (i.e., together with user-written Python code such as unit tests), and last how DTT can be used as a stand-alone tool that provides the same functionality without requiring the users to implement their tests using Python code.

2.8.1 The Table Class

The Drawn Table abstraction is implemented by the *Table* class. To create an instance, a name for the table must be given as well as a `str` with the Drawn Table. Further, a `str` representing `NULL` can optionally be given as well as a prefix to be used for variables (see *Variables*). The Drawn Table is then parsed by the following rules: The first row (called the "header") contains `name:type` pairs for each column with each pair surrounded by vertical pipes. After a type, `(pk)` can be specified to make a column (part of) the primary key. `UNIQUE` and `NOT NULL` are also supported and must be defined in the same manner as a primary key. If multiple constraints are defined for one column, they must be separated by a comma. Foreign keys are also supported and will be explained later. A valid header is, e.g., `| bid:int (pk) | title:text | genre:text |`. If the table should hold any data, the header must be followed by a delimiter line containing only vertical pipes, spaces, and dashes (`| -- | -- |`) and then each row follows on a line of its own. 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. A Drawn Table is also a valid table in GitHub Flavored Markdown. An example is given below.

```
import pygrametl.drawntabletesting as dtt

conn = dtt.connectionwrapper()

table = dtt.Table("book", """
| bid:int (pk) | title:text            | genre:text |
| ------------ | --------------------- | ---------- |
| 1            | Unknown               | Unknown    |
| 2            | Nineteen Eighty-Four  | Novel      |
| 3            | Calvin and Hobbes One | Comic      |
| 4            | Calvin and Hobbes Two | Comic      |
| 5            | The Silver Spoon      | Cookbook   |""")
table.ensure()
```

Alternatively, a Drawn Table's rows can be loaded from an external source by providing either a path to a file or an `iterable` to the constructor's `loadFrom` parameter. The file must contain a Drawn Table without a header and the `iterable` must yield `dicts` mapping from column names to values. Data can thus be loaded from files, databases, etc. at the cost of the test not being self-contained.

After a *Table* instance is created, its *ensure()* method can be invoked. 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). The *reset()* creates and fills the table even if it already exists, while *create()* creates the table without inserting any data into it. Finally, the SQL statement generated by the *Table* instance can be retrieved using the methods *getSQLToCreate()* and *getSQLToInsert()*. 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. It is thus a good choice to use for testing ETL flows during development. Another RDBMS can be used by calling *drawntabletesting.connectionwrapper()* with a PEP 249 connector.

Multiple different tables in the database can be represented using multiple instances of *Table*. In such situations, foreign keys constraints are often required. In DTT, foreign keys are defined in the same manner as the other constraints and require that users specify `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 `book` and `genre` can be seen below. All foreign key constraints are enforced by the RDMBS managing the test database.

```
import pygrametl.drawntabletesting as dtt

conn = dtt.connectionwrapper()

genre = dtt.Table("genre", """
| gid:int (pk) | genre:text |
```

(continues on next page)

(continued from previous page)

```
| ------------ | ---------- |
| 1            | Unknown    |
| 2            | Novel      |
| 3            | Comic      |
| 4            | Cookbook   |""")

book = dtt.Table("book", """
| bid:int (pk) | title:text             | gid:int (fk genre(gid)) |
| ------------ | ---------------------- | ----------------------- |
| 1            | Unknown                | 1                       |
| 2            | Nineteen Eighty-Four   | 2                       |
| 3            | Calvin and Hobbes One  | 3                       |
| 4            | Calvin and Hobbes Two  | 3                       |
| 5            | The Silver Spoon       | 4                       |""")
```

Table instances are immutable once created. Typically, the postcondition is, however, similar to the precondition except for a few added or updated rows. In DTT it is simple to create a new *Table* instance from an existing one by using the + operator.

```
newtable1 = book + "| 6 | Metro 2033 | 2 |" + "| 7 | Metro 2034 | 2 |"
```

A new instance is also created when one of the rows is updated. This is done by calling the *update()* method. For example, the first row in *table* can be changed with the line:

```
newtable2 = book.update(0, "| -1 | Unknown | -1 |")
```

Note that a new instance of *Table* is not represented in the test database unless its *ensure()* method is invoked. By making *Table* instances immutable and creating new instances when they are modified, it becomes very easy to 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. This is done using the method *additions()*. For example, a test case where the ETL flow is executed for the new rows is shown below.

```
def test_canInsertIntoBookDimensionTable(self):
    expected = table + "| 6 | Metro 2033 | 2 |" \
                     + "| 7 | Metro 2034 | 2 |"
    newrows = expected.additions()
    etl.executeETLFlow(newrows)
    expected.assertEqual()
```

For the code above, `expected` defines how the user expects the database state to become, but it is not the DTT framework that puts the database in this state. The database is modified by the ETL flow invoked by the user-provided `etl.executeETLFlow(newrows)` on Line 5. This method could, e.g., spawn a new process in which the user's ETL tool runs. It is thus *not* a requirement that the user's ETL flow is implemented in Python despite the tests being so. Using these features, DTT makes it simple to define the state of a database before a test is executed, and the rows the ETL flow should load. However, for the automatic test to be of any use, it is necessary to validate that the state of the database after the ETL flow has finished. This is done using assertions as shown on Line 6.

2.8.2 Assertions

DTT offers multiple assertions to check the state of a database table. At the moment, the methods *assertEqual()*, *assertDisjoint()*, and *assertSubset()* are implemented in DTT. When *assertEqual()* is called as shown above, DTT verifies that the table in the test database contains the expected rows (and only those) and if not, raises an `AssertionError` and provides an easy-to-read explanation of why the test failed as shown below.

```
AssertionError: book's rows differ from the rows in the database.
Drawn Table:
  | bid:int (pk) | title:text            | genre:text |
  | ------------ | --------------------- | ---------- |
  | 1            | Unknown               | Unknown    |
  | 2            | Nineteen Eighty-Four  | Novel      |
  | 3            | Calvin and Hobbes One | Comic      |
  | 4            | Calvin and Hobbes Two | Comic      |
  | 5            | The Silver Spoon      | Cookbook   |

Database Table:
  | bid:int (pk) | title:text            | genre:text |
  | ------------ | --------------------- | ---------- |
  | 1            | Unknown               | Unknown    |
  | 2            | Nineteen Eighty-Four  | Novel      |
  | 3            | Calvin and Hobbes One | Comic      |
  | 4            | Calvin and Hobbes Two | Cookbook   |
  | 5            | The Silver Spoon      | Cookbook   |

Violations:
  | bid:int (pk) | title:text            | genre:text |
  | ------------ | --------------------- | ---------- |
E | 4            | Calvin and Hobbes Two | Comic      |
  |              |                       |            |
D | 4            | Calvin and Hobbes Two | Cookbook   |
```

In this example, the part of the ETL flow loading the `book` table contains a bug. The *Table* instance in the test specifies that the dimension should contain a row for unknown books and four rows with known books (see the expected state in the top of the output). However, the user's ETL code wrongly added `Calvin and Hobbes Two` as a `Cookbook` instead of as a `Comic` (see the middle table in the output). To help the user quickly identify exactly what rows do not match, DTT prints the rows violating the assertion which for equality is the difference between the two drawn table and the database table (bottom). The expected rows (i.e., those in the *Table* instance) are prefixed by an `E` and the rows in the database table are prefixed by a `D`. The detailed information provided by *assertEqual()* can be disabled, by setting the optional parameter `verbose` to `False`. Note that the orders of the rows are allowed to differ between the Drawn Table and the database table without causing the test to fail.

When *assertDisjoint()* is called on a *Table* instance, it is asserted that none of the *Table*'s rows are present in the database table. In this way it is also possible to assert 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. When *assertSubset()* is called, it is asserted that all the *Table*'s rows are present in the database table which, however, may contain more rows which the user then does not have to specify. *assertSubset()* makes it easy to define a small set of rows that can be compared to a table with so many rows that they cannot be effectively embedded in the test itself. For example, it can easily be used to test if the leap day `2020-02-29` exists in the time dimension.

When compared to a table in the database, a *Table* instance does not have to contain all of the database table's columns. However, only the state of the included columns will be compared. This is useful for excluding columns for which the user does not know the state or which do not matter in the test, like an automatically generated primary key or audit information such as a timestamp.

2.8.3 Variables

In some cases specific cells must be equal across different database tables, but the exact values are unknown or do not matter. A prominent example is when foreign keys are used. In DTT this is easy to state using variables. A variable has a name prefixed by $ and can be used in any cell of a Drawn Table. The prefix can be changed by passing an argument to `variableprefix` in *Table*'s constructor. DTT checks if the cells with the same variable contain the same values in the database and fails the test if not. The code snippet below shows an example of how to use variables to test that foreign keys are assigned correctly.

```
import pygrametl.drawntabletesting as dtt

conn = dtt.connectionwrapper()

genre = dtt.Table("genre", """
| gid:int (pk)  | genre:text |
| ------------- | ---------- |
| $1            | Novel      |
| $2            | Comic      |""")

book = dtt.Table("book", """
| bid:int (pk) | title:text             | gid:int (fk genre(gid)) |
| ------------ | ---------------------- | ----------------------- |
| 1            | Nineteen Eighty-Four   | $1                      |
| 2            | Calvin and Hobbes One  | $2                      |
| 3            | Calvin and Hobbes Two  | $2                      |""")
```

Here it is stated that the `gid` for `Nineteen Eighty-Four` in `book` must match the `gid` for `Novel` in `genre`, while the `gid` for `Calvin and Hobbes One` and `Calvin and Hobbes Two` in `book` must match the `gid` for `Comic` in `genre`. If the variables with the same name do not have matching values, the errors shown below are raised.

```
...
AssertionError: Ambiguous values for $1; genre(row 0, column 0 gid) is 1 and book(row 0,␣
↪column 2 gid) is 2
...
```

This error message is an excerpt from the output of a test case where `genre` and `book` had their IDs defined in different orders. In this case, the foreign key constraints were satisfied although `Nineteen Eighty-Four` (wrongly) was referencing the genre `comic`. Thus, variables can test parts of the ETL flow which cannot be verified by foreign keys as the latter only ensure that a value is present.

Another example of using variables is shown below. Here the user verifies that in a type-2 Slowly Changing Dimension, the timestamp set for `validto` matches `validfrom` for the new version of the member. Thus, variables can be used to efficiently test automatically generated values are correct. It is also possible to specify that the value of a cell 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. In the example below, the actual values of the primary key column are not taken into consideration. `$_!` is a stricter version of `$_` which disallows `NULL`.

```
import pygrametl.drawntabletesting as dtt

conn = dtt.connectionwrapper()

address = dtt.Table("address", """
| aid:int (pk) | dept:text | location:text           | validfrom:date | validto:date |
| ------------ | --------- | ----------------------- | -------------- | ------------ |
```

(continues on next page)

(continued from previous page)

```
| $_           | CS        | Fredrik Bajers Vej 7    | 1990-01-01     | $1           |
| $_           | CS        | Selma Lagerløfs Vej 300 | $1             | NULL         |""
↪")
```

The methods *ensure()* and *reset()* may not be called on a Drawn Table where any variables are used (this will raise an error). This effectively means that variables can only be used when the postcondition is specified. The reason is that DTT does not know which concrete values to insert into the database for variables if they are used in preconditions.

2.8.4 Tooling Support

A key benefit of DTT is the ability for users to effectively understand the preconditions and postconditions of a test due to the visual representation provided by the Drawn Tables. However, to gain the full benefit of Drawn Tables, their columns must be aligned across rows as their content otherwise becomes much more difficult to read. A very poorly formatted Drawn Table can be seen below.

```
| bid:int (pk)    | title:text       | genre:text |
| ---------------- |
| 1     | Unknown    | Unknown |
| 2 | Nineteen Eighty-Four | Novel     |
| 3     | Calvin and Hobbes One     | Comic |
| 4 | Calvin and Hobbes Two     | Comic |
| 5        | The Silver Spoon | Cookbook |
```

It is clear from this example that poor formatting makes a Drawn Table harder to read. However, as properly formatting each Drawn Table can be tedious, DTT provides the script `formattable.py` that automates this task. The script is designed to be interfaced with extensible text editors so users can format a Drawn Table by simply placing the cursor anywhere on a Drawn Table and executing the script. An automatically formatted version of the Drawn Table from above can be seen below, and it is clear that this version of the Drawn Table is much easier to read.

```
| bid:int (pk) | title:text            | genre:text |
| ------------ | --------------------- | ---------- |
| 1            | Unknown               | Unknown    |
| 2            | Nineteen Eighty-Four  | Novel      |
| 3            | Calvin and Hobbes One | Comic      |
| 4            | Calvin and Hobbes Two | Comic      |
| 5            | The Silver Spoon      | Cookbook   |
```

The following two functions demonstrate how `formattable.py` can be integrated with GNU Emacs and Vim/NeoVim, respectively. However, `formattable.py` is editor agnostic and the functions are simply intended as examples.

GNU Emacs

```
(defun dtt-align-table ()
  "Format the Drawn Table at point using an external Python script."
  (interactive)
  (save-buffer)
  (shell-command
   (concat "python3 formattable.py " (buffer-file-name)
           " " (number-to-string (line-number-at-pos))))
  (revert-buffer :ignore-auto :noconfirm))
```

Vim and NeoVim

```
function! DTTAlignTable()
    write
    call system("python3 formattable.py " . expand('%:p') . " " . line('.'))
    edit!
endfunction
```

2.8.5 Drawn Table Testing as a Python Package

Using the presented constructs, users can efficiently define preconditions and postconditions to test each part of their ETL flows. DTT thus supports creation of tests during development, e.g., using test-driven development (TDD). A full example using both DTT and Python's `unittest` module is shown below.

When using `unittest`, a class must be defined for each set of tests. It is natural to group tests for a dimension into a class such that they can share a Drawn Table defining the precondition. A class using DTT to test the ETL flow for the book dimension is defined on Line 1. It inherits from `unittest.TestCase` as required by `unittest`. Two methods are then overridden `setUpClass()` and `setUp()`.

```
import unittest
import pygrametl.drawntabletesting as dtt


class BookStateTest(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.cw = dtt.connectionwrapper()
        cls.initial = dtt.Table("book", """
        | bid:int (pk) | title:text            | genre:text |
        | ------------ | --------------------- | ---------- |
        | 1            | Unknown               | Unknown    |
        | 2            | Nineteen Eighty-Four  | Novel      |
        | 3            | Calvin and Hobbes One | Comic      |
        | 4            | The Silver Spoon      | Cookbook   |""")

    def setUp(self):
        self.initial.reset()

    def test_insertNew(self):
        expected = self.initial + "| 5 | Calvin and Hobbes Two | Comic |"
        newrows = expected.additions()
        etl.executeETLFlow(self.cw, newrows)
        expected.assertEqual()

    def test_insertExisting(self):
        row = {'bid': 6, 'book': 'Calvin and Hobbes One', 'genre': 'Comic'}
        etl.executeETLFlow(self.cw, [row])
        self.initial.assertEqual()
```

The method `setUpClass()` is executed before the tests (methods starting with `test_`) in the class are executed. The method requests a database connection from DTT on Line 4 and defines a Drawn Table with the initial state of the dimension in Line 5. By creating them in `setUpClass()`, 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 the execution order of the tests, the book table in the database is reset before each test by `setUp()`. Then on Line 15 and Line 21, the tests are implemented as separate methods. `test_insertNew()` tests that a row that currently does not exist in book is inserted

correctly, while `test_insertExisting()` ensures that an already existing row does not become duplicated. In this example, both of these tests invoke the user's ETL flow by calling the user-defined method `executeETLFlow()`. As stated, the ETL flow may be implemented in Python, another programming language, or any other program.

2.8.6 Drawn Table Testing as a Stand-Alone Tool

DTT can also be used without doing any programming. To enable this, DTT provides a program with a command-line interface named `dttr` (for DTT Runner). Internally, `dttr` uses the DTT module described above. `dttr` uses test files, which have the `.dtt` suffix, to specify preconditions and/or postconditions. A test file only contains Drawn Tables but not any Python code. However, a configuration file named `config.py` can be created in the same folders as the `.dtt` files to define PEP 249 connections (i.e., in addition to the default in-memory SQlite database) and data sources (support for CSV and SQL is provided by `dttr`) for use in the tests. An example of a test file is given below. This file only contains one precondition (i.e., a Drawn Table with a name, but without an assert, on the first line) on Line 1–4 and one postcondition (i.e., a Drawn Table with both a name and an assert on the first line) on Line 6–13). This structure is, however, not a requirement as a `.dtt` file can contain any number of preconditions and/or postconditions.

```
book
| bid:int (pk) | title:text            | genre:text |
| ------------ | --------------------- | ---------- |
| 1            | Unknown               | Unknown    |

book, equal
| bid:int (pk) | title:text            | genre:text |
| ------------ | --------------------- | ---------- |
| 1            | Unknown               | Unknown    |
| 2            | Nineteen Eighty-Four  | Novel      |
| 3            | Calvin and Hobbes One | Comic      |
| 4            | Calvin and Hobbes Two | Comic      |
| 5            | The Silver Spoon      | Cookbook   |
```

To specify a precondition, first the name of the table must be given; in the example above that is `book`. As `dttr` uses the DTT module internally, it uses an in-memory SQLite database as the test database by default. Additional databases can be added by assigning PEP 249 connections to variables in the configuration file. To use a connection from the configuration file, the table name must be followed by an `@` sign and then the name of the connection to use for this table, e.g., `book@targetdw`. After the table name, a Drawn Table must be specified (Lines 2–4 in the example above). Like for any other Drawn Table, the header must be given first, then the delimiter, and last the rows. To mark the end of the precondition, an empty line is specified (Line 5).

To specify a postcondition, a table name must be given first. The table name must then followed by a comma and the name of the assertion to use as shown in Line 6 in the example. The table name for the postcondition is `book` like for the precondition, but they may also be different. For example, the precondition could define the initial state for `inputdata@sourcedb` and the postcondition could define the expected state for `book@targetdw`. As already mentioned, the name of the table to use for the postcondition is followed by a comma and the assertion to use, i.e., `equal` in this example. One can also use the other assertions in DTT: `disjoint` and `subset`. Finally in Line 7–13 the actual Drawn Table is given in the same way as for the precondition. The Drawn Table in the postcondition may also use variables. Note that a test does not require both a precondition and a postcondition, both are optional. It is thus, e.g., possible to create a test file where no precondition is set, but the postcondition still is asserted after executing the ETL flow. Also, as stated, a `.dtt` file can contain any number of preconditions and postconditions.

For tests that require more data than what is feasible to embed directly in a Drawn Table, data in an external file or database can be added to a Drawn Table by specifying an external data source as its last line. For example, by adding the line `csv bookdata.csv ,` the contents of the CSV file `bookdata.csv` is added to the Drawn Table with `,` used as field separator, in addition to any rows drawn as part of the Drawn Table. By adding `sql oltp SELECT bid, title, genre FROM book` as the last line, all rows of the table `book` from the PEP 249 connection `oltp` are added to

the Drawn Table. This is also extensible through the configuration file such that support for other sources of data, e.g., XML or a NoSQL DBMS like MongoDB can be added. This is done by creating a function in the configuration file. If, for example, the line `xml teacher 8` is found in a `.dtt` file, `dttr` looks for the function `xml` in the configuration file and executes it with the arguments `'teacher'` and `'8'`.

`dttr` can be invoked from the command line as shown below. Note that the ETL program to test and its arguments simply are given to `dttr` as arguments (`--etl ...`). Thus, any ETL program can be invoked.

```
$ ./dttr.py --etl "python3 myetl --loaddim book"
```

When executed, `dttr` by default looks for all `.dtt` test files in the current working directory, but optional arguments allow the user to select which files to consider (see `dttr -h` for more information). `dttr` then reads all relevant test files. Then the preconditions from these files are set. This is done using the DTT's *ensure* method such that each table is created and its data is inserted if necessary. If a table with the given name already exists and has differing content, an error will be raised and the table will not be updated. After the preconditions have been set, the ETL flow is started. How to execute the ETL flow is specified using the `--etl` flag as shown above. When the ETL flow has finished, all postconditions are asserted and any violation raises an error. If multiple occurrences of the same variable have different values, an error will also be raised, no matter if the variables are in the same or different `.dtt` files. It is thus, 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.

CHAPTER

THREE

API

3.1 pygrametl

A package for creating Extract-Transform-Load (ETL) programs in Python.

The package contains a number of classes for filling fact tables and dimensions (including snowflaked and slowly changing dimensions), classes for extracting data from different sources, classes for defining 'steps' in an ETL flow, and convenient functions for often-needed ETL functionality.

The package's modules are:

- datasources for access to different data sources
- tables for giving easy and abstracted access to dimension and fact tables
- parallel for parallelizing ETL operations
- JDBCConnectionWrapper and jythonmultiprocessing for support of Jython
- aggregators for aggregating data
- steps for defining steps in an ETL flow
- FIFODict for providing a dict with a limited size and where elements are removed in first-in first-out order

class pygrametl.**ConnectionWrapper**(*connection*, *stmtcachesize=1000*, *paramstyle=None*, *copyintonew=False*)

Bases: `object`

Provide a uniform representation of different database connection types.

All Dimensions and FactTables communicate with the data warehouse using a ConnectionWrapper. In this way, the code for loading the DW does not have to care about which parameter format is used.

pygrametl's code uses the 'pyformat' but the ConnectionWrapper performs translations of the SQL to use 'named', 'qmark', 'format', or 'numeric' if the user's database connection needs this. Note that the translations are simple and naive. Escaping as in %%(name)s is not taken into consideration. These simple translations are enough for pygrametl's code which is the important thing here; we're not trying to make a generic, all-purpose tool to get rid of the problems with different parameter formats. It is, however, possible to disable the translation of a statement to execute such that 'problematic' statements can be executed anyway.

Create a ConnectionWrapper around the given PEP 249 connection

If no default ConnectionWrapper already exists, the new ConnectionWrapper is set as the default.

Arguments:

- connection: An open PEP 249 connection to the database

- stmtcachesize: A number deciding how many translated statements to cache. A statement needs to be translated when the connection does not use 'pyformat' to specify parameters. When 'pyformat' is used and copyintonew == False, stmtcachesize is ignored as no statements need to be translated.
- paramstyle: A string holding the name of the PEP 249 connection's paramstyle. If None, pygrametl will try to find the paramstyle automatically (an AttributeError can be raised if that fails).
- copyintonew: A boolean deciding if a new mapping only holding the needed arguments should be created when a statement is executed. Some drivers require this.

close()

Close the connection to the database,

commit()

Commit the transaction.

cursor()

Return a cursor object. Optional method.

execute(*stmt*, *arguments=None*, *namemapping=None*, *translate=True*)

Execute a statement.

Arguments:

- stmt: the statement to execute
- arguments: a mapping with the arguments (default: None)
- namemapping: a mapping of names such that if stmt uses %(arg)s and namemapping[arg]=arg2, the value arguments[arg2] is used instead of arguments[arg]
- translate: decides if translation from 'pyformat' to the undlying connection's format should take place. Default: True

executemany(*stmt*, *params*, *translate=True*)

Execute a sequence of statements.

fetchalltuples()

Return all result tuples

fetchmanytuples(*cnt*)

Return cnt result tuples.

fetchone(*names=None*)

Return one result row (i.e. dict).

fetchonetuple()

Return one result tuple.

getunderlyingmodule()

Return a reference to the underlying connection's module.

This is done by considering the connection's __class__'s __module__ string from right to left (e.g., 'a.b.c', 'a.b', 'a') and looking for the attributes 'paramstyle' and 'connect' in the possible modules

resultnames()

rollback()

Rollback the transaction.

rowcount()

Return the size of the result.

rowfactory(*names=None*)

Return a generator object returning result rows (i.e. dicts).

setasdefault()

Set this ConnectionWrapper as the default connection.

pygrametl.**copy**(*row*, ***renaming*)

Create a copy of a dictionary, but allow renamings.

Arguments:

- row the dictionary to copy
- **renaming allows renamings to be specified in the form newname=oldname meaning that in the result, oldname will be renamed to newname. The key oldname must exist in the row argument, but it can be assigned to several newnames in the result as in x='repeated', y='repeated'.

pygrametl.**datereader**(*dateattribute*, *parsingfunction=ymdparser*)

Return a function that converts a certain dict member to a datetime.date

When setting, fromfinder for a tables.SlowlyChangingDimension, this method can be used for generating a function that picks the relevant dictionary member from each row and converts it.

Arguments:

- dateattribute: the attribute the generated function should read
- parsingfunction: the parsing function that converts the string to a datetime.date

pygrametl.**datespan**(*fromdate*, *todate*, *fromdateincl=True*, *todateincl=True*, *key='dateid'*, *strings={'date': '%Y-%m-%d', 'monthname': '%B', 'weekday': '%A'}*, *ints={'day': '%d', 'month': '%m', 'year': '%Y'}*, *expander=None*)

Return a generator yielding dicts for all dates in an interval.

Arguments:

- fromdate: The lower bound for the date interval. Should be a datetime.date or a YYYY-MM-DD formatted string.
- todate: The upper bound for the date interval. Should be a datetime.date or a YYYY-MM-DD formatted string.
- fromdateincl: Decides if fromdate is included. Default: True
- todateincl: Decides if todate is included. Default: True
- key: The name of the attribute where an int (YYYYMMDD) that uniquely identifies the date is stored. Default: 'dateid'.
- strings: A dict mapping attribute names to formatting directives (as those used by strftime). The returned dicts will have the specified attributes as strings. Default: {'date':'%Y-%m-%d', 'monthname':'%B', 'weekday':'%A'}
- ints: A dict mapping attribute names to formatting directives (as those used by strftime). The returned dicts will have the specified attributes as ints. Default: {'year':'%Y', 'month':'%m', 'day':'%d'}
- expander: A callable f(date, dict) that is invoked on each created dict. Not invoked if None. Default: None

pygrametl.**datetimereader**(*datetimeattribute*, *parsingfunction=ymdhmsparser*)

Return a function that converts a certain dict member to a datetime

When setting, fromfinder for a tables.SlowlyChangingDimension, this method can be used for generating a function that picks the relevant dictionary member from each row and converts it.

Arguments:

- datetimeattribute: the attribute the generated function should read
- parsingfunction: the parsing function that converts the string to a datetime.datetime

pygrametl.**endload**()

Signal to all Dimension and FactTable objects that all data is loaded.

pygrametl.**getbool**(*value*, *default=None*, *truevalues={True, '1', 't', 'true', 'True'}*, *falsevalues={False, 'false', 'False', 'f', '0'}*)

Convert a given value to True, False, or a default value.

If the given value is in the given truevalues, True is returned. If the given value is in the given falsevalues, False is returned. Otherwise, the default value is returned.

pygrametl.**getdate**(*targetconnection*, *ymdstr*, *default=None*)

Convert a string of the form 'yyyy-MM-dd' to a Date object.

The returned Date is in the given targetconnection's format.

Arguments:

- targetconnection: a ConnectionWrapper whose underlying module's Date format is used
- ymdstr: the string to convert
- default: The value to return if the conversion fails

pygrametl.**getdbfriendlystr**(*value*, *nullvalue='None'*)

Covert a value into a string that can be accepted by a DBMS.

None values are converted into the value of the argument nullvalues (default: 'None'). Bools are converted into '1' or '0' (instead of 'True' or 'False' as str would do). Other values are currently just converted by means of str.

pygrametl.**getdefaulttargetconnection**()

Return the default target connection

pygrametl.**getfloat**(*value*[, *default*]) → float(value) if possible, else default.

pygrametl.**getint**(*value*[, *default*]) → int(value) if possible, else default.

pygrametl.**getlong**(*value*[, *default*]) → long(value) if possible, else default.

pygrametl.**getstr**(*value*[, *default*]) → str(value) if possible, else default.

pygrametl.**getstrippedstr**(*value*, *default=None*)

Convert given value to a string and use .strip() on the result.

If the conversion fails, the given default value is returned.

pygrametl.**getstrornullvalue**(*value*, *nullvalue='None'*)

Convert a given value different from None to a string.

If the given value is None, nullvalue (default: 'None') is returned.

pygrametl.**gettimestamp**(*targetconnection*, *ymdhmsstr*, *default=None*)

Converts a string of the form 'yyyy-MM-dd HH:mm:ss' to a Timestamp.

The returned Timestamp is in the given targetconnection's format.

Arguments:

- targetconnection: a ConnectionWrapper whose underlying module's Timestamp format is used
- ymdhmsstr: the string to convert
- default: The value to return if the conversion fails

pygrametl.**getvalue**(*row*, *name*, *mapping={}*)

If name in mapping, return row[mapping[name]], else return row[name].

pygrametl.**getvalueor**(*row*, *name*, *mapping={}*, *default=None*)

Return the value of name from row using a mapping and a default value.

pygrametl.**keepasis**(*s*)

pygrametl.**now**(*ignoredtargetconn=None*, *ignoredrow=None*, *ignorednamemapping=None*)

Return the time of the first call this method as a datetime.datetime.

pygrametl.**project**(*atts*, *row*, *renaming={}*)

Create a new dictionary with a subset of the attributes.

Arguments:

- atts is a sequence of attributes in row that should be copied to the new result row.
- row is the original dictionary to copy data from.
- renaming is a mapping of names such that for each k in atts, the following holds:
  - If k in renaming then result[k] = row[renaming[k]].
  - If k not in renaming then result[k] = row[k].
  - renaming defaults to {}

pygrametl.**rename**(*row*, *renaming*)

Rename keys in a dictionary.

For each (oldname, newname) in renaming.items(): rename row[oldname] to row[newname].

pygrametl.**renamefromto**(*row*, *renaming*)

Rename keys in a dictionary.

For each (oldname, newname) in renaming.items(): rename row[oldname] to row[newname].

pygrametl.**renametofrom**(*row*, *renaming*)

Rename keys in a dictionary.

For each (newname, oldname) in renaming.items(): rename row[oldname] to row[newname].

pygrametl.**rowfactory**(*source*, *names*, *close=True*)

Generate dicts with key values from names and data values from source.

The given source should provide A) fetchmany returning the next set of rows, B) next() or fetchone() returning a tuple, or C) fetchall() returning a sequence of tuples. For each tuple, a dict is constructed such that the i'th element in names maps to the i'th value in the tuple.

If close=True (the default), close will be called on source after fetching all tuples.

pygrametl.**setdefaults**(*row*, *attributes*, *defaults=None*)

Set default values for attributes not present in a dictionary.

Default values are set for "missing" values, existing values are not updated.

Arguments:

- row is the dictionary to set default values in
- **attributes is either**
  - A) a sequence of attribute names in which case defaults must be an equally long sequence of these attributes default values or
  - B) a sequence of pairs of the form (attribute, defaultvalue) in which case the defaults argument should be None
- defaults is a sequence of default values (see above)

pygrametl.**today**(*ignoredtargetconn=None*, *ignoredrow=None*, *ignorednamemapping=None*)

Return the date of the first call this method as a datetime.date object.

pygrametl.**tolower**(*s*)

pygrametl.**toupper**(*s*)

pygrametl.**ymdhmsparser**(*ymdhmsstr*)

Convert an input with a string representation of the form 'yyyy-MM-dd HH:mm:ss' or a datetime.datetime to a datetime.datetime.

If the input is None, the return value is also None. If the input is a datetime.datetime, it is returned. Else the input is cast to str and quotes are stripped before it is split into the different parts needed to create a datetime.datetime.

pygrametl.**ymdparser**(*ymdstr*)

Convert an input with a string representation of the form 'yyyy-MM-dd' or a datetime.date or a datetime.datetime to a datetime.date.

If the input is None, the return value is also None. If the input is a datetime.date, it is returned. If the input is a datetime.datetime, its date is returned. Else the input is cast to str and quotes are stripped before it is split into the different parts needed to create a datetime.date.

3.2 datasources

This module holds classes that can be used as data soures. Note that it is easy to create other data sources: A data source must be iterable and provide dicts that map from attribute names to attribute values.

pygrametl.datasources.**BackgroundSource**

alias of *ProcessSource*

pygrametl.datasources.**CSVSource**

alias of `DictReader`

class pygrametl.datasources.**CrossTabbingSource**(*source*, *rowvaluesatt*, *colvaluesatt*, *values*, *aggregator=None*, *nonevalue=0*, *sortrows=False*)

Bases: `object`

A source that produces a crosstab from another source

Arguments:

- source: the data source to pull data from
- rowvaluesatt: the name of the attribute that holds the values that appear as rows in the result
- colvaluesatt: the name of the attribute that holds the values that appear as columns in the result
- values: the name of the attribute that holds the values to aggregate
- aggregator: the aggregator to use (see pygrametl.aggregators). If not given, pygrametl.aggregators.Sum is used to sum the values
- nonevalue: the value to return when there is no data to aggregate. Default: 0
- sortrows: A boolean deciding if the rows should be sorted. Default: False

class pygrametl.datasources.**DynamicForEachSource**(*seq*, *callee*)

Bases: object

A source that for each given argument creates a new source that will be iterated by this source.

For example, useful for directories where a CSVSource should be created for each file.

The user must provide a function that when called with a single argument, returns a new source to iterate. A DynamicForEachSource instance can be given to several ProcessSource instances.

Arguments:

- seq: a sequence with the elements for each of which a unique source must be created. the elements are given (one by one) to callee.
- callee: a function f(e) that must accept elements as those in the seq argument. the function should return a source which then will be iterated by this source. the function is called once for every element in seq.

class pygrametl.datasources.**FilteringSource**(*source*, *filter=<class 'bool'>*)

Bases: object

A source that applies a filter to another source

Arguments:

- source: the source to filter
- filter: a callable f(row). If the result is a True value, the row is passed on. If not, the row is discarded. Default: bool, i.e., Python's standard boolean conversion which removes empty rows.

class pygrametl.datasources.**HashJoiningSource**(*src1*, *key1*, *src2*, *key2*)

Bases: object

A class for equi-joining two data sources.

Arguments:

- src1: the first source. This source is iterated row by row.
- key1: the attribute of the first source to use in the join
- src2: the second source. The rows of this source are all loaded into memory.
- key2: the attriubte of the second source to use in the join.

pygrametl.datasources.**JoiningSource**

alias of *HashJoiningSource*

class pygrametl.datasources.**MappingSource**(*source*, *callables*)

Bases: object

A class for iterating a source and applying a function to each column.

Arguments:

- source: A data source
- callables: A dict mapping from attribute names to functions to apply to these names, e.g. type casting {'id':int, 'salary':float}

class pygrametl.datasources.**MergeJoiningSource**(*src1*, *key1*, *src2*, *key2*)

Bases: object

A class for merge-joining two sorted data sources

Arguments:

- src1: a data source
- key1: the attribute to use from src1
- src2: a data source
- key2: the attribute to use from src2

class pygrametl.datasources.**PandasSource**(*dataFrame*)

Bases: object

A source for iterating a Pandas DataFrame and cast each row to a dict.

Arguments:

- dataFrame: A Pandas DataFrame

class pygrametl.datasources.**ProcessSource**(*source*, *batchsize=500*, *queuesize=20*)

Bases: object

A class for iterating another source in a separate process

Arguments:

- source: the source to iterate
- batchsize: the number of rows passed from the worker process each time it passes on a batch of rows. Must be positive. Default: 500
- queuesize: the maximum number of batches that can wait in a queue between the processes. 0 means unlimited. Default: 20

class pygrametl.datasources.**RoundRobinSource**(*sources*, *batchsize=500*)

Bases: object

A source that reads sets of rows from sources in round robin-fashion

Arguments:

- sources: a sequence of data sources
- batchsize: the amount of rows to read from a data source before going to the next data source. Must be positive (to empty a source before going to the next, use UnionSource)

class pygrametl.datasources.**SQLSource**(*connection*, *query*, *names=()*, *initsql=None*, *cursorarg=None*, *parameters=None*)

Bases: object

A class for iterating the result set of a single SQL query.

Arguments:

- connection: the PEP 249 connection to use. NOT a ConnectionWrapper!
- query: the query that generates the result
- names: names of attributes in the result. If not set, the names from the database are used. Default: ()
- initsql: SQL that is executed before the query. The result of this initsql is not returned. Default: None.
- cursorarg: if not None, this argument is used as an argument when the connection's cursor method is called. Default: None.
- parameters: if not None, this sequence or mapping of parameters will be sent when the query is executed.

class pygrametl.datasources.**TransformingSource**(*source*, **transformations*)

Bases: object

A source that applies functions to the rows from another source

Arguments:

- source: a data source
- *transformations: the transformations to apply. Must be callables of the form func(row) where row is a dict. Will be applied in the given order.

class pygrametl.datasources.**TypedCSVSource**(*f*, *casts*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)

Bases: DictReader

A class for iterating a CSV file and type cast the values.

Arguments:

- f: An iterable object such as as file. Passed on to csv.DictReader
- casts: A dict mapping from attribute names to functions to apply to these names, e.g., {'id':int, 'salary':float}
- fieldnames: Passed on to csv.DictReader
- restkey: Passed on to csv.DictReader
- restval: Passed on to csv.DictReader
- dialect: Passed on to csv.DictReader
- *args: Passed on to csv.DictReader
- **kwds: Passed on to csv.DictReader

next()

class pygrametl.datasources.**UnionSource**(**sources*)

Bases: object

A source to union other sources (possibly with different types of rows). All rows are read from the 1st source before rows are read from the 2nd source and so on (to interleave the rows, use a RoundRobinSource)

Arguments:

- *sources: The sources to union in the order they should be used.

3.3 tables

This module contains classes for looking up rows, inserting rows and updating rows in dimensions and fact tables. Rows are represented as dictionaries mapping between attribute names and attribute values.

Note that named arguments should be used when instantiating classes. This improves readability and guards against errors in case of future API changes.

Many of the methods take an optional 'namemapping' argument which is explained here, but not repeated in the documentation for the individual methods: Consider a method m which is given a row r and a namemapping n. Assume that the method m uses the attribute a in r (i.e., r[a]). If the attribute a is not in the namemapping, m will just use r[a] as expected. But if the attribute a is in the namemapping, the name a is mapped to another name and the other name is used. That means that m then uses r[n[a]]. This is practical if attribute names in the considered rows and DW tables differ. If, for example, data is inserted into an order dimension in the DW that has the attribute order_date, but the source data uses the attribte name date, we can use a name mapping from order_date to date: dim.insert(row=..., namemapping={'order_date':'date'})

class pygrametl.tables.**AccumulatingSnapshotFactTable**(*name*, *keyrefs*, *otherrefs*, *measures=()*, *ignorenonerefs=True*, *ignorenonemeasures=True*, *factexpander=None*, *targetconnection=None*)

Bases: *FactTable*

A class for accessing and updating an accumulating fact table in the DW. Facts in an accumulating fact table can be updated. The class does no caching and all lookups and updates are sent to the DBMS (and an index on the keyrefs should thus be considered).

Arguments:

- name: the name of the fact table in the DW
- keyrefs: a sequence of attribute names that constitute the primary key of the fact tables. This is a subset of the dimension references and these references are not allowed to be updated.
- otherrefs: a sequence of dimension references that can be updated.
- measures: a possibly empty sequence of measure names. Default: ()
- ignorenonerefs: A flag deciding if None values for attributes in otherrefs are ignored when doing an update. If True (default), the existing value in the database will not be overwritten by a None.
- ignorenonemeasures: A flag deciding if None values for attributes in measures are ignored when doing an update. If True (default), the existing value in the database will not be overwritten by a None.
- factexpander: A function(row, namemapping, set of names of updated attributes). This function is called by the ensure method before it calls the update method if a row has been changed. This is, e.g., practical if lag measures should be computed before the row in the fact table is updated. The function should make its changes directly on the passed row.
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.

ensure(*row*, *namemapping={}*)

Lookup the given row. If that fails, insert it. If found, see if values for attributes in otherrefs or measures have changed and update the found row if necessary (note that values for attributes in keyrefs are not allowed to change). If an update is necessary and a factexpander is defined, the row will first be updated with any missing otherrefs/measures and the factexpander will be run on it. Return nothing.

Arguments:

- row: the row to insert or update if needed. Must contain the keyrefs attributes. For missing attributes from otherrefs and measures, the value is set to None if the row has to be inserted.
- namemapping: an optional namemapping (see module's documentation)

update(*row*, *namemapping={}*)

class pygrametl.tables.**BasePartitioner**(*parts*)

Bases: object

A base class for partitioning between several parts.

See also DimensionPartitioner and FactTablePartitioner.

addpart(*part*)

Add a part

droppart(*part=None*)

Drop a part. If an argument is given, it must be a part of the patitioner and it will then be removed. If no argument is given, the first part is removed.

endload()

Call endload on all parts

getpart(*row*, *namemapping={}*)

Find the part that should handle the given row. The provided implementation in BasePartitioner does only use round robin partitioning, but subclasses apply other methods

parts()

Return the parts the partitioner works on

class pygrametl.tables.**BatchFactTable**(*name*, *keyrefs*, *measures=()*, *batchsize=10000*, *usemultirow=False*, *targetconnection=None*)

Bases: *FactTable*

A class for accessing a fact table in the DW. This class performs performs insertions in batches.

Arguments:

- name: the name of the fact table in the DW
- keyrefs: a sequence of attribute names that constitute the primary key of the fact tables (i.e., the dimension references)
- measures: a possibly empty sequence of measure names. Default: ()
- batchsize: an int deciding many insert operations should be done in one batch. Default: 10000
- usemultirow: load batches with an INSERT INTO name VALUES statement instead of executemany(). WARNING: single quotes are automatically escaped. Other forms of sanitization must be manually performed.
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.

property awaitingrows

Return the amount of rows awaiting to be loaded into the table

endload()

Finalize the load.

class pygrametl.tables.**BulkDimension**(*name*, *key*, *attributes*, *bulkloader*, *lookupatts=()*, *idfinder=None*, *defaultidvalue=None*, *rowexpander=None*, *cachefullrows=False*, *fieldsep='\\t'*, *rowsep='\\n'*, *nullsubst=None*, *tempdest=None*, *bulksize=500000*, *usefilename=False*, *strconverter=pygrametl.getdbfriendlystr*, *encoding=None*, *dependson=()*, *targetconnection=None*)

Bases: _BaseBulkloadable, *CachedDimension*

A class for accessing a dimension table. Does caching and bulk loading.

Unlike CachedBulkDimension, this class always caches all dimension data.

The class caches all dimension members in memory. Newly inserted dimension members are also put into the cache. The class does not INSERT new dimension members into the underlying database table immediately when insert or ensure is invoked. Instead, the class does bulk loading of new members. When a certain amount of new dimension members have been inserted (configurable through __init__'s bulksize argument), a user-provided bulkloader method is called.

Calls of lookup and ensure will only use the cache and does not invoke any database operations. It is also possible to use the update and getbyvals methods, but calls of these will invoke the bulkloader first (and performance can degrade). If the dimension table's full rows are cached (by setting __init__'s cachefullrow argument to True), a call of getbykey will only use the cache, but if cachefullrows==False (which is the default), the bulkloader is again invoked first.

We assume that the DB doesn't change or add any attribute values that are cached. For example, a DEFAULT value in the DB or automatic type coercion can break this assumption.

Arguments:

- name: the name of the dimension table in the DW
- key: the name of the primary key in the DW
- attributes: a sequence of the attribute names in the dimension table. Should not include the name of the primary key which is given in the key argument.
- bulkloader: A method m(name, attributes, fieldsep, rowsep, nullsubst, tempdest) that is called to load data from a temporary file into the DW. The argument "attributes" is a list of the names of the columns to insert values into and show the order in which the attribute values appear in the temporary file. The rest of the arguments are similar to those arguments with identical names that are described below. The argument "tempdest" can, however, be 1) a string with a filename or 2) a file object. This is determined by the usefilename argument (see below).
- lookupatts: A subset of the attributes that uniquely identify a dimension members. These attributes are thus used for looking up members. If not given, it is assumed that lookupatts = attributes
- idfinder: A function(row, namemapping) -> key value that assigns a value to the primary key attribute based on the content of the row and namemapping. If not given, it is assumed that the primary key is an integer, and the assigned key value is then the current maximum plus one.
- defaultidvalue: An optional value to return when a lookup fails. This should thus be the ID for a preloaded "Unknown" member.
- rowexpander: A function(row, namemapping) -> row. This function is called by ensure before insertion if a lookup of the row fails. This is practical if expensive calculations only have to be done for rows that are not already present. For example, for a date dimension where the full date is used for looking up rows, a rowexpander can be set such that week day, week number, season, year, etc. are only calculated for dates that are not already represented. If not given, no automatic expansion of rows is done.
- cachefullrows: a flag deciding if full rows should be cached. If not, the cache only holds a mapping from lookupattributes to key values. Default: False.

- fieldsep: a string used to separate fields in the temporary file. Default: '\t'
- rowsep: a string used to separate rows in the temporary file. Default: '\n'
- nullsubst: an optional string used to replace None values. If nullsubst=None, no substitution takes place. Default: None
- tempdest: a file object or None. If None a named temporary file is used. Default: None
- bulksize: an int deciding the number of rows to load in one bulk operation. Default: 500000
- usefilename: a value deciding if the file should be passed to the bulkloader by its name instead of as a file-like object. This is, e.g., necessary when the bulk loading is invoked through SQL (instead of directly via a method on the PEP249 driver). It is also necessary if the bulkloader runs in another process. Default: False
- strconverter: a method m(value, nullsubst) -> str to convert values into strings that can be written to the temporary file and eventually bulkloaded. Default: pygrametl.getdbfriendlystr
- encoding: a string with the encoding to use. If None, locale.getpreferredencoding() is used. This argument is ignored under Python 2! Default: None
- dependson: a sequence of other bulkloadble tables that should be loaded before this instance does bulkloading. Default: ()
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.

getbykey(*keyvalue*)

Lookup and return the row with the given key value.

If no row is found in the dimension table, the function returns a row where all values (including the key) are None.

insert(*row*, *namemapping={}*)

Insert the given row. Return the new key value.

Arguments:

- row: the row to insert. The dict is not updated. It must contain all attributes, and is allowed to contain more attributes than that. Key is not required to be present but will be added using idfinder if missing.
- namemapping: an optional namemapping (see module's documentation)

class pygrametl.tables.**BulkFactTable**(*name*, *keyrefs*, *measures*, *bulkloader*, *fieldsep='\\t'*, *rowsep='\\n'*, *nullsubst=None*, *tempdest=None*, *bulksize=500000*, *usefilename=False*, *strconverter=pygrametl.getdbfriendlystr*, *encoding=None*, *dependson=()*)

Bases: _BaseBulkloadable

Class for addition of facts to a fact table. Reads are not supported.

Arguments:

- name: the name of the fact table in the DW
- keyrefs: a sequence of attribute names that constitute the primary key of the fact tables (i.e., the dimension references)
- measures: a possibly empty sequence of measure names.
- bulkloader: A method m(name, attributes, fieldsep, rowsep, nullsubst, tempdest) that is called to load data from a temporary file into the DW. The argument "attributes" is the combination of keyrefs and measures (i.e., a list of the names of the columns to insert values into) and show the order in which the attribute values appear in the temporary file. The rest of the arguments are similar to those arguments with identical

names that are given to BulkFactTable.__init__ as described here. The argument "tempdest" can, however, be 1) a string with a filename or 2) a file object. This is determined by the usefilename argument to BulkFactTable.__init__ (see below).

- fieldsep: a string used to separate fields in the temporary file. Default: '\t'
- rowsep: a string used to separate rows in the temporary file. Default: '\n'
- nullsubst: an optional string used to replace None values. If nullsubst=None, no substitution takes place. Default: None
- tempdest: a file object or None. If None a named temporary file is used. Default: None
- bulksize: an int deciding the number of rows to load in one bulk operation. Default: 500000
- usefilename: a value deciding if the file should be passed to the bulkloader by its name instead of as a file-like object. This is, e.g., necessary when the bulk loading is invoked through SQL (instead of directly via a method on the PEP249 driver). It is also necessary if the bulkloader runs in another process (for example, when if the BulkFactTable is wrapped by a DecoupledFactTable and invokes the bulkloader on a shared connection wrapper). Default: False
- strconverter: a method m(value, nullsubst) -> str to convert values into strings that can be written to the temporary file and eventually bulkloaded. Default: pygrametl.getdbfriendlystr
- encoding: a string with the encoding to use. If None, locale.getpreferredencoding() is used. This argument is ignored under Python 2! Default: None
- dependson: a sequence of other bulkloadble tables that should be bulkloaded before this instance does bulkloading (e.g., if the fact table has foreign keys to some bulk-loaded dimension table). Default: ()

class pygrametl.tables.**CachedBulkDimension**(*name*, *key*, *attributes*, *bulkloader*, *lookupatts=()*, *idfinder=None*, *defaultidvalue=None*, *rowexpander=None*, *usefetchfirst=False*, *cachefullrows=False*, *fieldsep='\\t'*, *rowsep='\\n'*, *nullsubst=None*, *tempdest=None*, *bulksize=5000*, *cachesize=10000*, *usefilename=False*, *strconverter=pygrametl.getdbfriendlystr*, *encoding=None*, *dependson=()*, *targetconnection=None*)

Bases: _BaseBulkloadable, *CachedDimension*

A class for accessing a dimension table. Does caching and bulk loading.

Unlike BulkDimension, the cache size is configurable and lookups may thus lead to database operations.

The class caches dimension members in memory. Newly inserted dimension members are also put into the cache. The class does not INSERT new dimension members into the underlying database table immediately when insert or ensure is invoked. Instead, the class does bulk loading of new members. When a certain amount of new dimension members have been inserted (configurable through __init__'s bulksize argument), a user-provided bulkloader method is called.

It is also possible to use the update and getbyvals methods, but calls of these will invoke the bulkloader first (and performance can degrade). If the dimension table's full rows are cached (by setting __init__'s cachefullrow argument to True), a call of getbykey will only use the cache, but if cachefullrows==False (which is the default), the bulkloader is again invoked first.

We assume that the DB doesn't change or add any attribute values that are cached. For example, a DEFAULT value in the DB or automatic type coercion can break this assumption.

Arguments:

- name: the name of the dimension table in the DW
- key: the name of the primary key in the DW

- attributes: a sequence of the attribute names in the dimension table. Should not include the name of the primary key which is given in the key argument.
- bulkloader: A method m(name, attributes, fieldsep, rowsep, nullsubst, tempdest) that is called to load data from a temporary file into the DW. The argument "attributes" is a list of the names of the columns to insert values into and show the order in which the attribute values appear in the temporary file. The rest of the arguments are similar to those arguments with identical names that are described below. The argument "tempdest" can, however, be 1) a string with a filename or 2) a file object. This is determined by the usefilename argument (see below).
- lookupatts: A subset of the attributes that uniquely identify a dimension members. These attributes are thus used for looking up members. If not given, it is assumed that lookupatts = attributes
- idfinder: A function(row, namemapping) -> key value that assigns a value to the primary key attribute based on the content of the row and namemapping. If not given, it is assumed that the primary key is an integer, and the assigned key value is then the current maximum plus one.
- defaultidvalue: An optional value to return when a lookup fails. This should thus be the ID for a preloaded "Unknown" member.
- rowexpander: A function(row, namemapping) -> row. This function is called by ensure before insertion if a lookup of the row fails. This is practical if expensive calculations only have to be done for rows that are not already present. For example, for a date dimension where the full date is used for looking up rows, a rowexpander can be set such that week day, week number, season, year, etc. are only calculated for dates that are not already represented. If not given, no automatic expansion of rows is done.
- usefetchfirst: a flag deciding if the SQL:2008 FETCH FIRST clause is used when prefil is True. Depending on the used DBMS and DB driver, this can give significant savings wrt. to time and memory. Not all DBMSs support this clause yet. Default: False
- cachefullrows: a flag deciding if full rows should be cached. If not, the cache only holds a mapping from lookupattributes to key values. Default: False.
- fieldsep: a string used to separate fields in the temporary file. Default: '\t'
- rowsep: a string used to separate rows in the temporary file. Default: '\n'
- nullsubst: an optional string used to replace None values. If nullsubst=None, no substitution takes place. Default: None
- tempdest: a file object or None. If None a named temporary file is used. Default: None
- bulksize: an int deciding the number of rows to load in one bulk operation. Default: 5000
- cachesize: the maximum number of rows to cache. If less than or equal to 0, unlimited caching is used. Default: 10000
- usefilename: a value deciding if the file should be passed to the bulkloader by its name instead of as a file-like object. This is, e.g., necessary when the bulk loading is invoked through SQL (instead of directly via a method on the PEP249 driver). It is also necessary if the bulkloader runs in another process. Default: False
- strconverter: a method m(value, nullsubst) -> str to convert values into strings that can be written to the temporary file and eventually bulkloaded. Default: pygrametl.getdbfriendlystr
- encoding: a string with the encoding to use. If None, locale.getpreferredencoding() is used. This argument is ignored under Python 2! Default: None
- dependson: a sequence of other bulkloadble tables that should be loaded before this instance does bulkloading. Default: ()
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.

getbykey(*keyvalue*)

Lookup and return the row with the given key value.

If no row is found in the dimension table, the function returns a row where all values (including the key) are None.

insert(*row*, *namemapping={}*)

Insert the given row. Return the new key value.

Arguments:

- row: the row to insert. The dict is not updated. It must contain all attributes, and is allowed to contain more attributes than that. Key is not required to be present but will be added using idfinder if missing.
- namemapping: an optional namemapping (see module's documentation)

lookup(*row*, *namemapping={}*)

Find the key for the row with the given values.

Arguments:

- row: a dict which must contain at least the lookup attributes
- namemapping: an optional namemapping (see module's documentation)

class pygrametl.tables.**CachedDimension**(*name*, *key*, *attributes*, *lookupatts=()*, *idfinder=None*, *defaultidvalue=None*, *rowexpander=None*, *size=10000*, *prefill=False*, *cachefullrows=False*, *cacheoninsert=True*, *usefetchfirst=False*, *targetconnection=None*)

Bases: *Dimension*

A class for accessing a dimension. Does caching.

We assume that the DB doesn't change or add any attribute values that are cached. For example, a DEFAULT value in the DB or automatic type coercion can break this assumption.

Arguments:

- name: the name of the dimension table in the DW
- key: the name of the primary key in the DW
- attributes: a sequence of the attribute names in the dimension table. Should not include the name of the primary key which is given in the key argument.
- lookupatts: A subset of the attributes that uniquely identify a dimension members. These attributes are thus used for looking up members. If not given, it is assumed that lookupatts = attributes
- idfinder: A function(row, namemapping) -> key value that assigns a value to the primary key attribute based on the content of the row and namemapping. If not given, it is assumed that the primary key is an integer, and the assigned key value is then the current maximum plus one.
- defaultidvalue: An optional value to return when a lookup fails. This should thus be the ID for a preloaded "Unknown" member.
- rowexpander: A function(row, namemapping) -> row. This function is called by ensure before insertion if a lookup of the row fails. This is practical if expensive calculations only have to be done for rows that are not already present. For example, for a date dimension where the full date is used for looking up rows, a rowexpander can be set such that week day, week number, season, year, etc. are only calculated for dates that are not already represented. If not given, no automatic expansion of rows is done.
- size: the maximum number of rows to cache. If less than or equal to 0, unlimited caching is used. Default: 10000

- prefill: a flag deciding if the cache should be filled when initialized. Default: False
- cachefullrows: a flag deciding if full rows should be cached. If not, the cache only holds a mapping from lookupattributes to key values. Default: False.
- cacheoninsert: a flag deciding if the cache should be updated when insertions are done. Default: True
- usefetchfirst: a flag deciding if the SQL:2008 FETCH FIRST clause is used when prefil is True. Depending on the used DBMS and DB driver, this can give significant savings wrt. to time and memory. Not all DBMSs support this clause yet. Default: False
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.

lookup(*row*, *namemapping={}*)

Find the key for the row with the given values.

Arguments:

- row: a dict which must contain at least the lookup attributes
- namemapping: an optional namemapping (see module's documentation)

class pygrametl.tables.**DecoupledDimension**(*dim*, *returnvalues=True*, *consumes=()*, *attstoconsume=()*, *batchsize=500*, *queuesize=200*)

Bases: *Decoupled*

A Dimension-like class that enables parallelism by executing all operations on a given Dimension in a separate, dedicated process (that Dimension is said to be "decoupled").

Arguments:

- dim: the Dimension object to use in a separate process
- returnvalues: decides if return values from method calls on dim should be kept such that they can be fetched by the caller or another Decoupled instance
- consumes: a sequence of Decoupled objects from which to fetch returnvalues (that are used to replace FutureResults in arguments). Default: ()
- attstoconsume: a sequence of the attribute names in rows that should have FutureResults replaced by actual return values. Does not have to be given, but may improve performance when given. Default: ()
- batchsize: the size of batches (grouped method calls) transferred between the processes. NB: Large values do not necessarily give good performance Default: 500
- queuesize: the maximum amount of waiting batches. Infinite if less than or equal to 0. NB: Large values do not necessarily give good performance. Default: 200

endload()

Invoke endload on the decoupled Dimension in the separate process and return when all waiting method calls have been executed

ensure(*row*, *namemapping={}*)

Invoke ensure on the decoupled Dimension in the separate process

getbykey(*keyvalue*)

Invoke getbykey on the decoupled Dimension in the separate process

getbyvals(*row*, *namemapping={}*)

Invoke betbycals on the decoupled Dimension in the separate process

insert(*row*, *namemapping={}*)

Invoke insert on the decoupled Dimension in the separate process

lookup(*row*, *namemapping={}*)

Invoke lookup on the decoupled Dimension in the separate process

scdensure(*row*, *namemapping={}*)

Invoke scdensure on the decoupled Dimension in the separate process

class pygrametl.tables.**DecoupledFactTable**(*facttbl*, *returnvalues=True*, *consumes=()*, *attstoconsume=()*, *batchsize=500*, *queuesize=200*)

Bases: *Decoupled*

A FactTable-like class that enables parallelism by executing all operations on a given FactTable in a separate, dedicated process (that FactTable is said to be "decoupled").

Arguments:

- facttbl: the FactTable object to use in a separate process
- returnvalues: decides if return values from method calls on facttbl should be kept such that they can be fetched by the caller or another Decoupled instance
- consumes: a sequence of Decoupled objects from which to fetch returnvalues (that are used to replace FutureResults in arguments). Default: ()
- attstoconsume: a sequence of the attribute names in rows that should have FutureResults replaced by actual return values. Does not have to be given, but may improve performance when given. Default: ()
- batchsize: the size of batches (grouped method calls) transferred between the processes. NB: Large values do not necessarily give good performance Default: 500
- queuesize: the maximum amount of waiting batches. Infinite if less than or equal to 0. NB: Large values do not necessarily give good performance. Default: 200

endload()

Invoke endload on the decoupled FactTable in the separate process and return when all waiting method calls have been executed

ensure(*row*, *namemapping={}*)

Invoke ensure on the decoupled FactTable in the separate process

insert(*row*, *namemapping={}*)

Invoke insert on the decoupled FactTable in the separate process

lookup(*row*, *namemapping={}*)

Invoke lookup on the decoupled FactTable in the separate process

class pygrametl.tables.**Dimension**(*name*, *key*, *attributes*, *lookupatts=()*, *idfinder=None*, *defaultidvalue=None*, *rowexpander=None*, *targetconnection=None*)

Bases: `object`

A class for accessing a dimension. Does no caching.

Arguments:

- name: the name of the dimension table in the DW
- key: the name of the primary key in the DW
- attributes: a sequence of the attribute names in the dimension table. Should not include the name of the primary key which is given in the key argument.
- lookupatts: A subset of the attributes that uniquely identify a dimension members. These attributes are thus used for looking up members. If not given, it is assumed that lookupatts = attributes

- idfinder: A function(row, namemapping) -> key value that assigns a value to the primary key attribute based on the content of the row and namemapping. If not given, it is assumed that the primary key is an integer, and the assigned key value is then the current maximum plus one.
- defaultidvalue: An optional value to return when a lookup fails. This should thus be the ID for a preloaded "Unknown" member.
- rowexpander: A function(row, namemapping) -> row. This function is called by ensure before insertion if a lookup of the row fails. This is practical if expensive calculations only have to be done for rows that are not already present. For example, for a date dimension where the full date is used for looking up rows, a rowexpander can be set such that week day, week number, season, year, etc. are only calculated for dates that are not already represented. If not given, no automatic expansion of rows is done.
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.

endload()

Finalize the load.

ensure(*row*, *namemapping={}*)

Lookup the given row. If that fails, insert it. Return the key value.

If the lookup fails and a rowexpander was set when creating the instance, this rowexpander is called before the insert takes place.

Arguments:

- row: the row to lookup or insert. Must contain the lookup attributes. Key is not required to be present but will be added using idfinder if missing.
- namemapping: an optional namemapping (see module's documentation)

getbykey(*keyvalue*)

Lookup and return the row with the given key value.

If no row is found in the dimension table, the function returns a row where all values (including the key) are None.

getbyvals(*values*, *namemapping={}*)

Return a list of all rows with values identical to the given.

Arguments:

- values: a dict which must hold a subset of the tables attributes. All rows that have identical values for all attributes in this dict are returned.
- namemapping: an optional namemapping (see module's documentation)

insert(*row*, *namemapping={}*)

Insert the given row. Return the new key value.

Arguments:

- row: the row to insert. The dict is not updated. It must contain all attributes, and is allowed to contain more attributes than that. Key is not required to be present but will be added using idfinder if missing.
- namemapping: an optional namemapping (see module's documentation)

lookup(*row*, *namemapping={}*)

Find the key for the row with the given values.

Arguments:

- row: a dict which must contain at least the lookup attributes

- namemapping: an optional namemapping (see module's documentation)

update(*row*, *namemapping={}*)

Update a single row in the dimension table.

Arguments:

- row: a dict which must contain the key for the dimension. The row with this key value is updated such that it takes the value of row[att] for each attribute att which is also in row.
- namemapping: an optional namemapping (see module's documentation)

class pygrametl.tables.**DimensionPartitioner**(*parts*, *getbyvalsfromall=False*, *partitioner=None*)

Bases: *BasePartitioner*

A Dimension-like class that handles partitioning.

Partitioning is done between a number of Dimension objects called the parts. The class offers the interface of Dimensions (incl. scdensure from SlowlyChangingDimension). When a method is called, the corresponding method on one of the parts (chosen by a user-definable partitioner function) will be invoked. The parts can operate on a single physical dimension table or different physical tables.

Arguments:

- parts: a sequence of Dimension objects.
- getbyvalsfromall: determines if getbyvals should be answered by means of all parts (when getbyvalsfromall = True) or only the first part, i.e., parts[0] (when getbybalsfromall = False). Default: False
- partitioner: None or a callable p(dict) -> int where the argument is a dict mapping from the names of the lookupatts to the values of the lookupatts. The resulting int is used to determine which part a given row should be handled by. When partitioner is None, a default partitioner is used. This partitioner computes the hash value of each value of the lookupatts and adds them together.

ensure(*row*, *namemapping={}*)

Invoke ensure on the relevant Dimension part

getbykey(*keyvalue*)

Invoke getbykey on the relevant Dimension part

getbyvals(*values*, *namemapping={}*)

Invoke getbyvals on the first part or all parts (depending on the value of the instance's getbyvalsfromall)

getpart(*row*, *namemapping={}*)

Return the part that should handle the given row

insert(*row*, *namemapping={}*)

Invoke insert on the relevant Dimension part

lookup(*row*, *namemapping={}*)

Invoke lookup on the relevant Dimension part

scdensure(*row*, *namemapping={}*)

Invoke scdensure on the relevant Dimension part

update(*row*, *namemapping={}*)

Invoke update on the relevant Dimension part

class pygrametl.tables.**FactTable**(*name*, *keyrefs*, *measures=()*, *targetconnection=None*)

Bases: object

A class for accessing a fact table in the DW.

Arguments:

- name: the name of the fact table in the DW
- keyrefs: a sequence of attribute names that constitute the primary key of the fact tables (i.e., the dimension references)
- measures: a possibly empty sequence of measure names. Default: ()
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.

endload()

Finalize the load.

ensure(*row*, *compare=False*, *namemapping={}*)

Ensure that a fact is present (insert it if it is not already there).

Return True if a fact with identical values for keyrefs attributes was already present in the fact table; False if not.

Arguments:

- row: a dict at least containing the attributes of the fact table
- compare: a flag deciding if measure vales from a fact that was looked up are compared to those in the given row. If True and differences are found, a ValueError is raised. Default: False
- namemapping: an optional namemapping (see module's documentation)

insert(*row*, *namemapping={}*)

Insert a fact into the fact table.

Arguments:

- row: a dict at least containing values for all the fact table's attributes (both keys/references and measures).
- namemapping: an optional namemapping (see module's documentation)

lookup(*keyvalues*, *namemapping={}*)

Lookup a fact from the given key values. Return key and measure vals.

Return None if no fact is found.

Arguments:

- keyvalues: a dict at least containing values for all keys
- namemapping: an optional namemapping (see module's documentation)

class pygrametl.tables.**FactTablePartitioner**(*parts*, *partitioner=None*)

Bases: *BasePartitioner*

A FactTable-like class that handles partitioning.

Partitioning is done between a number of FactTable objects called the parts. The class offers the interface of FactTable. When a method is called, the corresponding method on one of the parts (chosen by a user-definable partitioner function) will be invoked. The parts can operate on a single physical fact table or different physical tables.

Arguments:

- parts: a sequence of FactTable objects.
- partitioner: None or a callable p(dict) -> int where the argument is a dict mapping from the names of the keyrefs to the values of the keyrefs. The resulting int is used to determine which part a given row should be handled by. When partitioner is None, a default partitioner is used. This partitioner computes the sum of all the keyrefs values.

ensure(*row*, *namemapping={}*)

Invoke ensure on the relevant part

getpart(*row*, *namemapping={}*)

Return the relevant part for the given row

insert(*row*, *namemapping={}*)

Invoke insert on the relevant part

lookup(*row*, *namemapping={}*)

Invoke lookup on the relevant part

class pygrametl.tables.**SlowlyChangingDimension**(*name*, *key*, *attributes*, *lookupatts*, *orderingatt=None*, *versionatt=None*, *fromatt=None*, *fromfinder=None*, *toatt=None*, *tofinder=None*, *minfrom=None*, *maxto=None*, *srcdateatt=None*, *srcdateparser=pygrametl.ymdparser*, *type1atts=()*, *cachesize=10000*, *prefill=False*, *idfinder=None*, *usefetchfirst=False*, *useorderby=True*, *targetconnection=None*)

Bases: *Dimension*

A class for accessing a slowly changing dimension of "type 2".

"Type 1" updates can also be applied for a subset of the attributes.

Caching is used. We assume that the DB doesn't change or add any attribute values that are cached. For example, a DEFAULT value in the DB or automatic type coercion can break this assumption.

Arguments:

- name: the name of the dimension table in the DW
- key: the name of the primary key in the DW
- attributes: a sequence of the attribute names in the dimension table. Should not include the name of the primary key which is given in the key argument, but should include versionatt, fromatt, and toatt.
- lookupatts: a sequence with a subset of the attributes that uniquely identify a dimension members. These attributes are thus used for looking up members.
- orderingatt: the name of the attribute used to identify the newest version. The version holding the greatest value is considered to be the newest. If orderingatt is None, versionatt is used. If versionatt is also None, toatt is used and NULL is considered as the greatest value. If orderingatt, versionatt,toatt are all None, an error is raised.
- versionatt: the name of the attribute holding the version number
- fromatt: the name of the attribute telling from when the version becomes valid. Not used if None. Default: None
- fromfinder: a function(targetconnection, row, namemapping) returning a value for the fromatt for a new version (the function is first used when it is determined that a new version must be added; it is not applied to determine this). If fromfinder is None and srcdateatt is also None, pygrametl.today is used as fromfinder. If fromfinder is None and srcdateatt is not None, pygrametl.datereader(srcdateatt, srcdateparser) is used.

In other words, if no date attribute and no special date function are given, new versions get the date of the current day. If a date attribute is given (but no date function), the date attribute's value is converted (by means of srcdateparser) and a new version gets the result of this as the date it is valid from. Default: None

- toatt: the name of the attribute telling until when the version is valid. Not used if None. Default: None
- tofinder: a function(targetconnection, row, namemapping) returning a value for the toatt. If not set, fromfinder is used (note that if fromfinder is None, it is set to a default function – see the comments about fromfinder. The possibly modified value is used here.) Default: None
- minfrom: the value to use for fromatt for the 1st version of a member if fromatt is not already set. If None, the value is found in the same way as for other new versions, i.e., as described for fromfinder. If fromatt should take the value NULL for the 1st version, set minfrom to a tuple holding a single element which is None: (None,). Note that minto affects the 1st version, not any following versions. Note also that if the member to insert already contains a value for fromatt, minfrom is ignored. Default: None.
- maxto: the value to use for toatt for new members. Default: None
- srcdateatt: the name of the attribute in the source data that holds a date showing when a version is valid from. The data is converted to a datetime by applying srcdateparser on it. If not None, the date attribute is also used when comparing a potential new version to the newest version in the DB. If None, the date fields are not compared. Default: None
- srcdateparser: a function that takes one argument (a date in the format scrdateatt has) and returns a datetime.datetime. If srcdateatt is None, srcdateparser is not used. Default: pygrametl.ymdparser (i.e., the default value is a function that parses a string of the form 'yyyy-MM-dd')
- type1atts: a sequence of attributes that should have type1 updates applied. Default: ()
- cachesize: the maximum size of the cache. 0 disables caching and values smaller than 0 allows unlimited caching
- prefill: decides if the cache should be prefilled with the newest versions. Default: False.
- idfinder: a function(row, namemapping) -> key value that assigns a value to the primary key attribute based on the content of the row and namemapping. If not given, it is assumed that the primary key is an integer, and the assigned key value is then the current maximum plus one.
- usefetchfirst: a flag deciding if the SQL:2008 FETCH FIRST clause is used when prefil is True. Depending on the used DBMS and DB driver, this can give significant savings wrt. to time and memory. Not all DBMSs support this clause yet. Default: False
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.
- useorderby: a flag deciding if ORDER BY is used in the SQL to select the newest version. If True, the DBMS thus does the sorting. If False, all versions are fetched and the highest version is found in Python. For some systems, this can lead to significant performance improvements. Default: True

closecurrent(*row*, *namemapping={}*, *end=datetime.date(2022, 12, 5)*)

Close the current version by setting its toatt if it is maxto.

The newest version will have its toatt set to the given end argument (default pygrametl.today(), i.e., the current date) only if, the value for the newest row's toatt currently is maxto. Otherwise no update will be done. If toatt is not defined an exception is raised.

Arguments:

- row: a dict which must contain at least the lookup attributes
- namemapping: an optional namemapping (see module's documentation)
- end: the value to set for the newest version. Default: The current date as given by pygrametl.today()

lookup(*row*, *namemapping={}*)

Find the key for the newest version with the given values.

Arguments:

- row: a dict which must contain at least the lookup attributes
- namemapping: an optional namemapping (see module's documentation)

scdensure(*row*, *namemapping={}*)

Lookup or insert a version of a slowly changing dimension member.

Note: Has side-effects on the given row.

Arguments:

- row: a dict containing the attributes for the member. key, versionatt, fromatt, and toatt are not required to be present but will be added (if defined).
- namemapping: an optional namemapping (see module's documentation)

class pygrametl.tables.**SnowflakedDimension**(*references*, *expectboguskeyvalues=False*)

Bases: `object`

A class for accessing a snowflaked dimension spanning several tables in the underlying database. Lookups and inserts are then automatically spread out over the relevant tables while the programmer only needs to interact with a single SnowflakedDimension instance.

Arguments:

- references: a sequence of pairs of Dimension objects [(a1,a2), (b1,b2), ...] meaning that a1 has a foreign key to a2 etc. a2 may itself be a sequence of Dimensions: [(a1, [a21, a22, ...]), (b1, [b21, b22, ...]), ...].

  The first element of the first pair (a1 in the example above) must be the dimension table representing the lowest level in the hierarchy (i.e., the dimension table the closest to the fact table).

  Each dimension must be reachable in a unique way (i.e., the given dimensions form a tree).

  A foreign key must have the same name as the primary key it references.

- expectboguskeyvalues: If expectboguskeyvalues is True, we allow a key that is used as lookup attribute in a lower level to hold a wrong value (which would typically be None). When ensure or insert is called, we find the correct value for the key in the higher level. If expectboguskeyvalues, we again try a lookup on the lower level after this. If expectboguskeyvalues is False, we move directly on to do an insert. Default: False

endload()

Finalize the load.

ensure(*row*, *namemapping={}*)

Lookup the given member. If that fails, insert it. Return key value.

If the member must be inserted, data is automatically inserted in all participating tables where (part of) the member is not already represented.

Key values for different levels may be added to the row. It is NOT guaranteed that key values for all levels exist in row afterwards.

Arguments:

- row: the row to lookup or insert. Must contain the lookup attributes. Key is not required to be present but will be added using idfinder if missing.

- namemapping: an optional namemapping (see module's documentation)

`getbykey`(*keyvalue*, *fullrow=False*)

Lookup and return the row with the given key value.

If no row is found in the dimension table, the function returns a row where all values (including the key) are None.

Arguments:

- keyvalue: the key value of the row to lookup
- fullrow: a flag deciding if the full row (with data from all tables in the snowflake) should be returned. If False, only data from the lowest level in the hierarchy (i.e., the table the closest to the fact table) is returned. Default: False

`getbyvals`(*values*, *namemapping={}*, *fullrow=False*)

Return a list of all rows with values identical to the given.

Arguments:

- values: a dict which must hold a subset of the tables attributes. All rows that have identical values for all attributes in this dict are returned.
- namemapping: an optional namemapping (see module's documentation)
- fullrow: a flag deciding if the full row (with data from all tables in the snowflake) should be returned. If False, only data from the lowest level in the hierarchy (i.e., the table the closest to the fact table) is returned. Default: False

`insert`(*row*, *namemapping={}*)

Insert the given member. If that fails, insert it. Return key value.

Data is automatically inserted in all participating tables where (part of) the member is not already represented. If nothing is inserted at all, a ValueError is raised.

Key values for different levels may be added to the row. It is NOT guaranteed that key values for all levels exist in row afterwards.

Arguments:

- row: the row to lookup or insert. Must contain the lookup attributes. Key is not required to be present but will be added using idfinder if missing.
- namemapping: an optional namemapping (see module's documentation)

`lookup`(*row*, *namemapping={}*)

Find the key for the row with the given values.

Arguments:

- row: a dict which must contain at least the lookup attributes which all must come from the root (the table closest to the fact table).
- namemapping: an optional namemapping (see module's documentation)

`scdensure`(*row*, *namemapping={}*)

Lookup or insert a version of a slowly changing dimension member.

> **Warning:** Still experimental!!! For now we require that only the root is a SlowlyChangingDimension.

Note: Has side-effects on the given row.

Arguments:

- row: a dict containing the attributes for the member. Key is not required to be present but will be added using idfinder if missing.
- namemapping: an optional namemapping (see module's documentation)

update(*row*, *namemapping={}*)

Update rows in the participating dimension tables.

If the key of a participating dimension D is in the given row, D.update(...) is invoked.

Note that this function is not good to use for updating a foreign key which here has the same name as the referenced primary key: The referenced table could then also get updated unless it is ensured that none of its attributes are present in the given row.

In other words, it is often better to use the update function directly on the Dimensions that should be updated.

Arguments:

- row: a dict. If the key of a participating dimension D is in the dict, D.update(...) is invoked.
- namemapping: an optional namemapping (see module's documentation)

class pygrametl.tables.**SubprocessFactTable**(*keyrefs*, *measures*, *executable*, *initcommand=None*, *endcommand=None*, *terminateafter=-1*, *fieldsep='\\t'*, *rowsep='\\n'*, *nullsubst=None*, *strconverter=pygrametl.getdbfriendlystr*, *buffersize=16384*)

Bases: object

Class for addition of facts to a subprocess.

The subprocess can, e.g., be a logger or bulkloader. Reads are not supported.

Note that a created instance can not be used when endload() has been called (and endload() is called from pygrametl.commit()).

Arguments:

- keyrefs: a sequence of attribute names that constitute the primary key of the fact table (i.e., the dimension references)
- measures: a possibly empty sequence of measure names. Default: ()
- executable: The subprocess to start.
- initcommand: If not None, this command is written to the subprocess before any data.
- endcommand: If not None, this command is written to the subprocess after all data has been written.
- terminateafter: If greater than or equal to 0, the subprocess is terminated after this amount of seconds after the pipe to the subprocess is closed.
- fieldsep: a string used to separate fields in the output sent to the subprocess. Default: '\t
- rowsep: a string used to separate rows in the output sent to the subprocess. Default: '\n'
- nullsubst: an optional string used to replace None values. If nullsubst=None, no substitution takes place. Default: None
- strconverter: a method m(value, nullsubst) -> str to convert values into strings that can be written to the subprocess. Default: pygrametl.getdbfriendlystr

endload()

Finalize the load.

insert(*row*, *namemapping={}*)

Insert a fact into the fact table.

Arguments:

- row: a dict at least containing values for the keys and measures.
- namemapping: an optional namemapping (see module's documentation)

class pygrametl.tables.**TypeOneSlowlyChangingDimension**(*name*, *key*, *attributes*, *lookupatts*, *type1atts=()*, *cachesize=10000*, *prefill=False*, *idfinder=None*, *usefetchfirst=False*, *cachefullrows=False*, *targetconnection=None*)

Bases: *CachedDimension*

A class for accessing a slowly changing dimension of "type 1".

Caching is used. We assume that the DB doesn't change or add any attribute values that are cached. For example, a DEFAULT value in the DB or automatic type coercion can break this assumption.

Arguments:

- name: the name of the dimension table in the DW
- key: the name of the primary key in the DW
- attributes: a sequence of the attribute names in the dimension table. Should not include the name of the primary key which is given in the key argument.
- lookupatts: A subset of the attributes that uniquely identify a dimension members. These attributes are thus used for looking up members.
- type1atts: A sequence of attributes that should have type1 updates applied, it cannot intersect with lookupatts. If not given, it is assumed that type1atts = attributes - lookupatts
- cachesize: the maximum number of rows to cache. If less than or equal to 0, unlimited caching is used. Default: 10000
- prefill: a flag deciding if the cache should be filled when initialized. Default: False
- idfinder: A function(row, namemapping) -> key value that assigns a value to the primary key attribute based on the content of the row and namemapping. If not given, it is assumed that the primary key is an integer, and the assigned key value is then the current maximum plus one.
- usefetchfirst: a flag deciding if the SQL:2008 FETCH FIRST clause is used when prefil is True. Depending on the used DBMS and DB driver, this can give significant savings wrt. to time and memory. Not all DBMSs support this clause yet. Default: False
- cachefullrows: a flag deciding if full rows should be cached. If not, the cache only holds a mapping from lookupattributes to key values, and from key to the type 1 slowly changing attributes. Default: False.
- targetconnection: The ConnectionWrapper to use. If not given, the default target connection is used.

scdensure(*row*, *namemapping={}*)

Lookup or insert a version of a slowly changing dimension member.

Note: Has side-effects on the given row.

Arguments:

- row: a dict containing the attributes for the table. It must contain all attributes if it is the first version of the row to be inserted, updates of existing rows need only contain lookupatts and a subset of type1atts as a missing type1atts is ignored and the existing value left as is in the database. Key is not required to be present but will be added using idfinder if missing.
- namemapping: an optional namemapping (see module's documentation)

pygrametl.tables.**definequote**(*quotechar*)

Defines the global quote function, for wrapping identifiers with quotes.

Arguments:

- quotechar: If None, do not wrap identifier. If a string, prepend and append quotechar to identifier. If a tuple of two strings, prepend with first element and append with last.

3.4 parallel

This module contains methods and classes for making parallel ETL flows. Note that this module in many cases will give better results with Jython (where it uses threads) than with CPython (where it uses processes).

class pygrametl.parallel.**Decoupled**(*obj*, *returnvalues=True*, *consumes=()*, *directupdatepositions=()*, *batchsize=500*, *queuesize=200*, *autowrap=True*)

Bases: object

shutdowndecoupled()

Let the Decoupled instance finish its tasks and stop it.

The Decoupled instance should not be used after this.

pygrametl.parallel.**createflow**(**functions*, ***options*)

Create a flow of functions running in different processes.

A Flow object ready for use is returned.

A flow consists of several functions running in several processes. A flow created by

flow = createflow(f1, f2, f3)

uses three processes. Data can be inserted into the flow by calling it as in flow(data). The argument data is then first processed by f1(data), then f2(data), and finally f3(data). Return values from f1, f2, and f3 are *not* preserved, but their side-effects are. The functions in a flow should all accept the same number of arguments (*args are also okay).

Internally, a Flow object groups calls together in batches to reduce communication costs (see also the description of arguments below). In the example above, f1 could thus work on one batch, while f2 works on another batch and so on. Flows are thus good to use even if there are many calls of relatively fast functions.

When no more data is to be inserted into a flow, it should be closed by calling its close method.

Data processed by a flow can be fetched by calling get/getall or simply iterating the flow. This can both be done by the process that inserted data into the flow or by another (possibly concurrent) process. All data in a flow should be fetched again as it otherwise will remain in memory .

Arguments:

- *functions: A sequence of functions of sequences of functions. Each element in the sequence will be executed in a separate process. For example, the argument (f1, (f2, f3), f4) leads to that f1 executes in

process-1, f2 and f3 execute in process-2, and f4 executes in process-3. The functions in the sequence should all accept the same number of arguments.

- **options: keyword arguments configuring details. The considered options are:
  - batchsize: an integer deciding how many function calls are "grouped together" before they are passed on between processes. The default is 500.
  - queuesize: an integer deciding the maximum number of batches that can wait in a JoinableQueue between two different processes. 0 means that there is no limit. The default is 25.

pygrametl.parallel.**endsplits**()

Wait for all splitpoints to finish

pygrametl.parallel.**getsharedsequencefactory**(*startvalue*, *intervallen=5000*)

Creates a factory for parallel readers of a sequence.

Returns a callable f. When f() is called, it returns a callable g. Whenever g(*args) is called, it returns a unique int from a sequence (if several g's are created, the order of the calls may lead to that the returned ints are not ordered, but they will be unique). The arguments to g are ignored, but accepted. Thus g can be used as idfinder for [Decoupled]Dimensions.

The different g's can be used safely from different processes and threads.

Arguments:

- startvalue: The first value to return. If None, 0 is assumed.
- intervallen: The amount of numbers that a single g from above can return before synchronization is needed to get a new amount. Default: 5000.

pygrametl.parallel.**shareconnectionwrapper**(*targetconnection*, *maxclients=10*, *userfuncs=()*)

Share a ConnectionWrapper between several processes/threads.

When Decoupled objects are used, they can try to update the DW at the same time. They can use several ConnectionWrappers to avoid race conditions, but this is not transactionally safe. Instead, they can use a "shared" ConnectionWrapper obtained through this function.

When a ConnectionWrapper is shared, it is executing in a separate process (or thread, in case Jython is used) and ensuring that only one operation takes place at the time. This is hidden from the users of the shared ConnectionWrapper. They see an interface similar to the normal ConnectionWrapper.

When this method is called, it returns a SharedConnectionWrapperClient which can be used as a normal ConnectionWrapper. Each process (i.e., each Decoupled object) should, however, get a unique SharedConnectionWrapperClient by calling copy() on the returned SharedConnectionWrapperClient.

Note that a shared ConnectionWrapper needs to hold the complete result of each query in memory until it is fetched by the process that executed the query. Again, this is hidden from the users.

It is also possible to add methods to a shared ConnectionWrapper when it is created. When this is done and the method is invoked, no other operation will modify the DW at the same time. If, for example, the functions foo and bar are added to a shared ConnectionWrapper (by passing the argument userfuncs=(foo, bar) to shareconnectionwrapper), the returned SharedConnectionWrapperClient will offer the methods foo and bar which when called will be running in the separate process for the shared ConnectionWrapper. This is particularly useful for user-defined bulk loaders as used by BulkFactTable:

def bulkload():
DBMS-specific code here. No other DW operation should take place concurrently

scw = shareconnectionwrapper(ConnectionWrapper(...), userfuncs=(bulkload,))

facttbl = BulkFact(..., bulkloader=scw.copy().bulkload)

Note: The SharedConnectionWrapper must be copied using .copy().

Arguments:

- targetconnection: a pygrametl ConnectionWrapper
- maxclients: the maximum number of concurrent clients. Default: 10
- userfuncs: a sequence of functions to add to the shared ConnectionWrapper. Default: ()

pygrametl.parallel.**splitpoint**(**arg*, ***kwargs*)

To be used as an annotation to make a function run in a separate process.

Each call of a @splitpoint annotated function f involves adding the request (and arguments, if any) to a shared queue. This can be relatively expensive if f only uses little computation time. The benefits from @splitpoint are thus best obtained for a function f which is time-consuming. To wait for all splitpoints to finish their computations, call endsplits().

@splitpoint can be used as in the following examples:

@splitpoint
def f(args):

The simplest case. Makes f run in a separate process. All calls of f will return None immediately and f will be invoked in the separate process.

@splitpoint()
def g(args):

With parentheses. Has the same effect as the previous example.

@splitpoint(output=queue, instances=2, queuesize=200)
def h(args):

With keyword arguments. It is not required that all of keyword arguments above are given.

Keyword arguments:

- output: If given, it should be a queue-like object (offering the .put(obj) method). The annotated function's results will then be put in the output
- instances: Determines how many processes should run the function. Each of the processes will have the value parallel.splitno set to a unique value between 0 (incl.) and instances (excl.).
- queuesize: Given as an argument to a multiprocessing.JoinableQueue which holds arguments to the annotated function while they wait for an idle process that will pass them on to the annotated function. The argument decides the maximum number of calls that can wait in the queue. 0 means unlimited. Default: 0

3.5 JDBCConnectionWrapper

This module holds a ConnectionWrapper that is used with a JDBC Connection. The module should only be used when running Jython.

class pygrametl.JDBCConnectionWrapper.**BackgroundJDBCConnectionWrapper**(*jdbcconn*, *stmtcachesize=20*)

Bases: object

Wrap a JDBC Connection and do all DB communication in the background.

All Dimension and FactTable communicate with the data warehouse using a ConnectionWrapper. In this way, the code for loading the DW does not have to care about which parameter format is used. This ConnectionWrapper is a special one for JDBC in Jython and does DB communication from a Thread.

Note: BackgroundJDBCConnectionWrapper is added for experiments. It is quite similar to JDBCConnectionWrapper and one of them may be removed.

Create a ConnectionWrapper around the given JDBC connection

Arguments:

- jdbcconn: An open JDBC Connection (not a PEP249 Connection)
- stmtcachesize: The maximum number of PreparedStatements kept open. Default: 20.

close()

Close the connection to the database,

commit()

Commit the transaction.

cursor()

Not implemented for this JDBC connection wrapper!

execute(*stmt*, *arguments=None*, *namemapping=None*, *ignored=None*)

Execute a statement.

Arguments:

- stmt: the statement to execute
- arguments: a mapping with the arguments. Default: None.
- namemapping: a mapping of names such that if stmt uses %(arg)s and namemapping[arg]=arg2, the value arguments[arg2] is used instead of arguments[arg]
- ignored: An ignored argument only present to accept the same number of arguments as ConnectionWrapper.execute

executemany(*stmt*, *params*, *ignored=None*)

Execute a sequence of statements.

Arguments:

- stmt: the statement to execute
- params: a sequence of arguments
- ignored: An ignored argument only present to accept the same number of arguments as ConnectionWrapper.executemany

fetchalltuples()

Return all result tuples

fetchmanytuples(*cnt*)

Return cnt result tuples.

fetchone(*names=None*)

Return one result row (i.e. dict).

fetchonetuple()

Return one result tuple.

getunderlyingmodule()

Return a reference to the underlying connection's module.

resultnames()

rollback()

Rollback the transaction.

rowcount()

Not implemented. Return 0. Should return the size of the result.

rowfactory(*names=None*)

Return a generator object returning result rows (i.e. dicts).

setasdefault()

Set this ConnectionWrapper as the default connection.

class pygrametl.JDBCConnectionWrapper.**JDBCConnectionWrapper**(*jdbcconn*, *stmtcachesize=20*)

Bases: object

Wrap a JDBC Connection.

All Dimension and FactTable communicate with the data warehouse using a ConnectionWrapper. In this way, the code for loading the DW does not have to care about which parameter format is used. This ConnectionWrapper is a special one for JDBC in Jython.

Create a ConnectionWrapper around the given JDBC connection.

If no default ConnectionWrapper already exists, the new ConnectionWrapper is set to be the default ConnectionWrapper.

Arguments:

- jdbcconn: An open JDBC Connection (not a PEP249 Connection)
- stmtcachesize: The maximum number of PreparedStatements kept open. Default: 20.

close()

Close the connection to the database,

commit()

Commit the transaction.

cursor()

Not implemented for this JDBC connection wrapper!

execute(*stmt*, *arguments=None*, *namemapping=None*, *ignored=None*)

Execute a statement.

Arguments:

- stmt: the statement to execute
- arguments: a mapping with the arguments. Default: None.
- namemapping: a mapping of names such that if stmt uses %(arg)s and namemapping[arg]=arg2, the value arguments[arg2] is used instead of arguments[arg]
- ignored: An ignored argument only present to accept the same number of arguments as ConnectionWrapper.execute

executemany(*stmt*, *params*, *ignored=None*)

Execute a sequence of statements.

Arguments:

- stmt: the statement to execute
- params: a sequence of arguments
- ignored: An ignored argument only present to accept the same number of arguments as ConnectionWrapper.executemany

fetchalltuples()

Return all result tuples

fetchmanytuples(*cnt*)

Return cnt result tuples.

fetchone(*names=None*)

Return one result row (i.e. dict).

fetchonetuple()

Return one result tuple.

getunderlyingmodule()

Return a reference to the underlying connection's module.

resultnames()

rollback()

Rollback the transaction.

rowcount()

Not implemented. Return 0. Should return the size of the result.

rowfactory(*names=None*)

Return a generator object returning result rows (i.e. dicts).

setasdefault()

Set this ConnectionWrapper as the default connection.

3.6 jythonmultiprocessing

A module for Jython emulating (a small part of) CPython's multiprocessing. With this, pygrametl can be made to use multiprocessing, but actually use threads when used from Jython (where there is no GIL).

class pygrametl.jythonmultiprocessing.**JoinableQueue**(*maxsize=0*)

Bases: *Queue*

close()

class pygrametl.jythonmultiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, ***, *daemon=None*)

Bases: Thread

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

property daemon

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

property name

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

pid = '<n/a>'

class pygrametl.jythonmultiprocessing.**Queue**(*maxsize=0*)

Bases: object

Create a queue object with a given maximum size.

If maxsize is <= 0, the queue size is infinite.

empty()

Return True if the queue is empty, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0 as a direct substitute, but be aware that either approach risks a race condition where a queue can grow before the result of empty() or qsize() can be used.

To create code that needs to wait for all queued tasks to be completed, the preferred technique is to use the join() method.

full()

Return True if the queue is full, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() >= n as a direct substitute, but be aware that either approach risks a race condition where a queue can shrink before the result of full() or qsize() can be used.

get(*block=True*, *timeout=None*)

Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until an item is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Empty exception if no item was available within that time. Otherwise ('block' is false), return an item if one is immediately available, else raise the Empty exception ('timeout' is ignored in that case).

get_nowait()

Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

join()

Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

put(*item*, *block=True*, *timeout=None*)

Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

put_nowait(*item*)

Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

qsize()

Return the approximate size of the queue (not reliable!).

task_done()

Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

class pygrametl.jythonmultiprocessing.**Value**

Bases: object

3.7 aggregators

A module with classes for aggregation. An Aggregator has two methods: process and finish.

process(group, val) is called to "add" val to the aggregation of the set of values identified by the value of group. The value in group (which could be any hashable type, also a tuple as ('A', 'B')) thus corresponds to the GROUP BY attributes in SQL.

finish(group, default) is called to get the final result for group. If no such results exists, default is returned.

class pygrametl.aggregators.**Aggregator**

Bases: object

finish(*group*, *default=None*)

process(*group*, *val*)

class pygrametl.aggregators.**Avg**

Bases: *Aggregator*

finish(*group*, *default=None*)

process(*group*, *val*)

class pygrametl.aggregators.**Count**

Bases: *SimpleAggregator*

process(*group*, *val*)

class pygrametl.aggregators.**CountDistinct**

Bases: *SimpleAggregator*

finish(*group*, *default=None*)

process(*group*, *val*)

class pygrametl.aggregators.**Max**

Bases: *SimpleAggregator*

process(*group*, *val*)

class pygrametl.aggregators.**Min**

Bases: *SimpleAggregator*

process(*group*, *val*)

class pygrametl.aggregators.**SimpleAggregator**

Bases: *Aggregator*

finish(*group*, *default=None*)

process(*group*, *val*)

class pygrametl.aggregators.**Sum**

Bases: *SimpleAggregator*

process(*group*, *val*)

3.8 steps

This module contains classes for making "steps" in an ETL flow. Steps can be connected such that a row flows from step to step and each step does something with the row.

class pygrametl.steps.**ConditionalStep**(*condition*, *whentrue*, *whenfalse=None*, *name=None*)

Bases: *Step*

A Step that redirects rows based on a condition.

Arguments:

- condition: A function f(row) that is evaluated for each row.
- whentrue: The next step to use if the condition evaluates to a true value. This argument should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on.
- whenfalse: The Step that rows are sent to when the condition evaluates to a false value. If None, the rows are silently discarded. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

class pygrametl.steps.**CopyStep**(*originaldest*, *copydest*, *deepcopy=False*, *name=None*)

Bases: *Step*

A Step that copies each row and passes on the copy and the original

Arguments:

- originaldest: The Step each given row is passed on to. This argument should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on.
- copydest: The Step a copy of each given row is passed on to. This argument can be 1) an instance of a Step or 2) the name of a step.
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None
- deepcopy: Decides if the copy should be deep or not. Default: False

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

class pygrametl.steps.**DimensionStep**(*dimension*, *keyfield=None*, *next=None*, *name=None*)

Bases: *Step*

A Step that performs ensure(row) on a given dimension for each row.

Arguments:

- dimension: the Dimension object to call ensure on.

- keyfield: the name of the attribute that in each row is set to hold the key value for the dimension member
- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

class pygrametl.steps.**GarbageStep**(*name=None*)

Bases: *Step*

A Step that does nothing. Rows are neither modified nor passed on.

Arguments:

- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

process(*row*)

Perform the Step's operation on the given row.

If the row is not explicitly redirected (see _redirect), it will be passed on the the next step if this has been set.

class pygrametl.steps.**MappingStep**(*targets*, *requiretargets=True*, *next=None*, *name=None*)

Bases: *Step*

A Step that applies functions to attributes in rows.

Arguments:

- targets: A sequence of (name, function) pairs. For each element, row[name] is set to function(row[name]) for each row given to the step.
- requiretargets: A flag that decides if a KeyError should be raised if a name from targets does not exist in a row. If True, a KeyError is raised, if False the missing attribute is ignored and not set. Default: True
- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

class pygrametl.steps.**PrintStep**(*next=None*, *name=None*)

Bases: *Step*

A Step that prints each given row.

Arguments:

- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

class pygrametl.steps.**RenamingFromToStep**(*renaming*, *next=None*, *name=None*)

Bases: *Step*

Step that performs renamings of attributes in rows.

Arguments:

- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None
- renaming: A dict with pairs (oldname, newname) which will by used by pygrametl.renamefromto to do the renaming
- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

pygrametl.steps.**RenamingStep**

alias of *RenamingFromToStep*

class pygrametl.steps.**RenamingToFromStep**(*renaming*, *next=None*, *name=None*)

Bases: *RenamingFromToStep*

Arguments:

- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None
- renaming: A dict with pairs (oldname, newname) which will by used by pygrametl.renamefromto to do the renaming
- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

class pygrametl.steps.**SCDimensionStep**(*dimension*, *next=None*, *name=None*)

Bases: *Step*

A Step that performs scdensure(row) on a given dimension for each row.

Arguments:

- dimension: the Dimension object to call ensure on.
- keyfield: the name of the attribute that in each row is set to hold the key value for the dimension member
- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

class pygrametl.steps.**SourceStep**(*source*, *next=None*, *name=None*)

Bases: *Step*

A Step that iterates over a data source and gives each row to the next step. The start method must be called.

Arguments:

- source: The data source. Must be iterable.
- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

start()

Start the iteration of the source's rows and pass them on.

class pygrametl.steps.**Step**(*worker=None*, *next=None*, *name=None*)

Bases: `object`

The basic class for steps in an ETL flow.

Arguments:

- worker: A function f(row) that performs the Step's operation. If None, self.defaultworker is used. Default: None
- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None

- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

classmethod getstep(*name*)

Return the Step instance with the given name

name()

Return the name of the Step instance

process(*row*)

Perform the Step's operation on the given row.

If the row is not explicitly redirected (see _redirect), it will be passed on the the next step if this has been set.

class pygrametl.steps.**ValueMappingStep**(*outputatt*, *inputatt*, *mapping*, *requireinput=True*, *defaultvalue=None*, *next=None*, *name=None*)

Bases: *Step*

A Step that Maps values to other values (e.g., DK -> Denmark)

Arguments:

- outputatt: The attribute to write the mapped value to in each row.
- inputatt: The attribute to map.
- mapping: A dict with the mapping itself.
- requireinput: A flag that decides if a KeyError should be raised if inputatt does not exist in a given row. If True, a KeyError will be raised when the attriubte is missing. If False, a the outputatt will be set to defaultvalue. Default: True
- defaultvalue: The default value to use when the mapping cannot be done. Default: None
- next: The default next step to use. This should be 1) an instance of a Step, 2) the name of a Step, or 3) None. If if is a name, the next step will be looked up dynamically each time. If it is None, no default step will exist and rows will not be passed on. Default: None
- name: A name for the Step instance. This is used when another Step (implicitly or explicitly) passes on rows. If two instanes have the same name, the name is mapped to the instance that was created the latest. Default: None

defaultworker(*row*)

Perform the Step's operation on the given row.

Inheriting classes should implement this method.

pygrametl.steps.**connectsteps**(**steps*)

Set a.next = b, b.next = c, etc. when given the steps a, b, c, ...

3.9 FIFODict

A simple mapping between keys and values, but with a limited capacity. When the max. capacity is reached, the first inserted key/value pair is deleted

pygrametl.FIFODict.**FIFODict**

alias of FIFODictOrderedDict

3.10 drawntabletesting

This module contains classes and functions for defining preconditions and postconditions for database state. The conditions can be used in unit tests to efficiently evaluate the expected database state.

class pygrametl.drawntabletesting.**Table**(*name*, *table*, *nullsubst='NULL'*, *variableprefix='$'*, *loadFrom=None*, *testconnection=None*)

Bases: object

A class representing a concrete database table.

Note that the asserts assume that the Table instance and the database table do not have duplicate rows, if they do the asserts raise an error.

Arguments:

- name: the name of the table in the database.
- table: the contents of the table as an ASCII drawing.
- nullsubst: a string that represents NULL in the drawn table.
- variableprefix: a string all variables must have as a prefix.
- loadFrom: additional rows to be loaded as a path to a file that continues the table argument or as an iterable producing dicts.
- testconnection: The connection wrapper to use for the unit tests. If None pygrametl's current default connection wrapper is used.

additions(*withKey=False*)

Return all rows added or updated since the original drawn table.

Arguments:

- withKey: if True the primary keys are included in the rows.

assertDisjoint(*verbose=True*)

Return a Boolean indicating if none of the rows in this object occur in the database table. If verbose=True an AssertionError is also raised if the row sets are not disjoint.

Arguments:

- verbose: if True an ASCII representation of the rows violating the assertion is printed as part of the AssertionError.

assertEqual(*verbose=True*)

Return a Boolean indicating if the rows in this object and the rows in the database table match. If verbose=True an AssertionError is also raised if the rows don't match.

Arguments:

- verbose: if True an ASCII representation of the rows violating the assertion is printed as part of the AssertionError.

assertSubset(*verbose=True*)

Return a Boolean stating if the rows in this object are a subset of those in the database table. If verbose=True an AssertionError is also raised if the rows are not a subset of those in the database table.

Arguments:

- verbose: if True an ASCII representation of the rows violating the assertion is printed as part of the AssertionError.

classmethod clear()

Drop all tables and variables without checking their contents.

create()

Create the table if it does not exist without adding any rows.

drop()

Drop the table in the database without checking the contents.

ensure()

Create the table if it does not exist, otherwise verify the rows.

If the table does exist but does not contain the expected set of rows an exception is raised to prevent overriding existing data.

getSQLToCreate()

Return a string of SQL that creates the table.

getSQLToInsert()

Return a string of SQL that inserts all rows into the table.

key()

Return the primary key.

For a simple primary key, the name is returned. For a composite primary key, all names are returned in a tuple.

reset()

Forcefully create a new table and add the provided rows.

update(*index*, *line*)

Create a new instance with the row specified by the index updated with the values included in the provided line.

Arguments:

- index: the index of the row to be updated.
- line: an ASCII representation of the new values.

pygrametl.drawntabletesting.**connectionwrapper**(*connection=None*)

Create a new connection wrapper for use with unit tests.

Arguments:

- connection: A PEP249 connection to use. If None (the default), a connection to a temporary SQLite in-memory database is created.