Resume

Dette projekt omhandler udviklingen af python-frameworket, SkiRaff, beregnet til automatiseret test af ETL-programmer, udviklet ved brug af pygrametl python-pakken. Frameworkets mål er at forsimple implementationen af tests uden særlig forøgelse af runtime. Jo nemmere det er at implementere tests, jo flere tests kan der blive udført. Hvilket kan forøge softwarens endelige kvalitet.

ETL står på engelsk for extraction, transformation og load. Betegnelsen refererer til systemer, der fører data fra en mængde datakilder over i et fælles datawarehouse(DW). Dataen kan blive ændret under overførsel. Sådanne systemer tillader, at data fra forskellige kilder kan blive lagret og analyseret sammen. Mange ETL-værktøjer fokuserer på, at disse systemer skal udvikles gennem en GUI. Men da eksperter oftest arbejder hurtigere ved brug af en API, stiller pygrametl sådan en til rådighed.

For at sikre kvaliteten af et ETL-system skal det testes. Værktøjer til ETL-testing er i dag oftest GUI-baserede. Dette stemmer ikke overens med pygrametls brug. I stedet kan man teste ETL-systemer manuelt, eller der kan anvendes mere generiske testværktøjer, men det er ikke er en optimal løsning. Dertil udvikler vi SkiRaff, en python-pakke, der skal hjælpe pygrametl-programmører med at teste deres programmer.

Ligesom med andre typer software-testing, anvender man test-cases til test af ETL-systemer. En test-case er en påstand om, hvordan et program skal fungere i en given situation. Når en case er blevet defineret, kan den implementeres, og tjekke at casen overholdes af programmet. Test-cases kan henvende sig til alle strukturelle niveauer i et ETL-system. Da vi antager at brugerne af pygrametl er eksperter, der kan anvende unit-tests, fokuserer vi med SkiRaff ikke på test ved komponent og integrations niveauerne. I stedet er SkiRaff beregnet til test på system niveauet. Mere nøjagtigt anvendes frameworket til datadreven source-to-target testing. Her testes et ETL-system igennem den data, som det skriver ind i et DW. Hvis fejlagtig data optræder i DW'et indikerer det fejl i ETL-systemet.

SkiRaff består af to større komponenter, DWPopulator-klassen og familien af Predicate-klasser. DWPopulator kører det ETL-system, som man ønsker at teste. Hertil populere DWPopulator et udvalgt DW, der kan anvendes til test. DWPopulator tillader også dynamisk udskiftning af kilder og DW inde i programmet før det køres. Dette gør det nemmere for brugere at ændre på den data, som der skal testes på. Efter at DW'et er blevet fyldt, testes det ved brug af predicate-klasserne. Hver predicate-klasse tillader test af en bestemt DW egenskab, som skal opfyldes. En instantiering af et predicate kan da ses som implementation af en specifik test-case. Predicaterne tillader tests for størrelse af tabeller, lighed mellem tabeller, functional dependency med mere. Efter udførsel af et predicate instance rapporteres der, hvorvidt den pågældende egenskab blev overholdt. Hvis antagelsen fejler rapporteres der også om rækker i DW'et, der ledte til udfaldet.

Efter udvikling af SkiRaff var tilendebragt evaluerede vi frameworket op imod manuel testing, hvor der anvendes SQLite queries igennem et python script. Med begge metoder skulle de samme egenskaber for et DW testes. Vi fandt her, at SkiRaff krævede langt færre program erklæringer. Samtidig viste der sig ikke en særlig forskel i udførselstid imellem de to metoder. Dog skal det også siges, at de test, som hver metode skulle udføre passede godt til vores predicates. Der findes højst sandsynligt test cases, hvor den større fleksibilitet af SQL vil være mere gavnlig. Vi føler dog, at SkiRaff's predicates giver den nødvendige testdækning påkrævet, og at det er nemmere at implementere end manuel testing.

Grouproom and group email: A6-323 (d608f16@cs.aau.dk)
Supervisor: Christian Thomsen (chr@cs.aau.dk)

SkiRaff, an ETL Testing Framework for pygrametl

Alexander Brandborg[1] (abran13@student.aau.dk), Arash Michael Sami Kjær[1] (ams13@...), Mathias Claus Jensen[1] (mcje13@...), Mikael Vind Mikkelsen[1] (mvmi12@...)

Abstract
The python package pygrametl allows developers to construct ETL systems through an API. We develop a testing framework, SkiRaff, which allows testers to evaluate programs developed using pygrametl. Our motivation is that no such tool has been developed for use with pygrametl. The framework is developed for both functional- and regression testing at the system level. It is based on source to target testing. Here an ETL system is tested by asserting about properties of the DW, which it populates. SkiRaff can be used to check whether assertions are upheld by such a DW. These assertions are implemented using SkiRaff's predicate classes. These allow testers to assert about DW properties relating to data loss and business rules. After development we evaluate SkiRaff against manual testing, where testers write SQL to test for properties. We found that both methods result in test implementations with the same runtime. Yet, while manual testing needed 110 statements for its implementation, SkiRaff needed only 11. This indicates a benefit for testers in migrating from manual testing to SkiRaff.

Keywords
Data Warehouse — ETL — Testing — SkiRaff — pygrametl

[1] *Student at the Department of Computer Science, Aalborg University, Aalborg, Denmark*

1. Introduction

To ensure that a piece of software can be expected to run at low risk of failure, it must be tested. Today we have specialized software that assist in the automatic execution of tests. This allows testers to focus more on what to test and less on test implementation. This leads to tests of a higher quality.

Some testing tools have a broad application, while other test tools have a more specific use. Per extension, there exists many tools exclusively used for testing *Extract-Transform-Load* (ETL) systems. These systems support the creation and update of *data warehouses* (DW), which are mainly used for business analysis. An ETL process will extract data from a set of sources, apply transformations to that data and load it into a DW. Often GUI-based tools are used, when developing ETL systems. The pygrametl python package, which is open source, allows coding of entire ETL systems instead. The main idea being that experts perform better when using an API rather than a GUI [1]. Our goal is to develop a testing framework, which can assist users of pygrametl in testing their ETL systems.

ETL tests may be set up manually using SQL. Yet, in the current market many different testing tools exists. These allow for the automation of tests. Testing often occurs by focusing on the DW, which the ETL system populates. QuerySurge[2] for example, focuses on comparing the data from sources to the data in the DW. The tool is built on the idea that data is often unaltered, as it passes through the ETL process. This means that column A in a source must be one-to-one mappable to column B in the DW. Testing of this type only requires the user to supply some simple mappings between sources and DW. As such, QuerySurge presents itself as a novice-friendly solution and allows for testing through a GUI. Yet, if mappings between source data and DW become more complex, testers will need to write SQL code to test. We determine that this kind of tool is not fitting for the users of pygrametl. As pygrametl users are expert users, who will not gain anything from the novice-friendly features of QuerySurge. The amount of code needed for testing may also be rather large, if there is a frequent need for SQL.

Testing may also occur through the comparison of tables within a populated DW with those defined by the tester. By constructing a table, testers assert how a DW table should look after load. The truth of this assertion can be found by comparing to the actual DW table. For this, tools such as AnyDBTest[3] may be used. AnyDBTest also allows testers to describe, what kind of comparison should be made. For example, we may assert that one table is the subset of another. The problem with this type of testing, is that it requires the setup of many user-defined tables. Again, a large amount of test code needs to be written. However, we do believe that table comparison is a powerful tool for ETL testing. Yet, comparison between tables should not be the only assertable property of a DW.

Having looked at some of the currently available tools, we find that they require a lot of test code to be written. To get a wider test coverage, we need to decrease the amount of code necessary for each test. We also worry that faults may be more prevalent to occur in the test code, as it grows larger. Such faults can undermine the usefulness and accuracy of test results.

To allow for automated testing of pygrametl programs, we develop *SkiRaff*. Our motivation for developing SkiRaff is that no other testing tool has been made with pygrametl in mind. As of writing, testers will have to test manually or use third-party tools that are either rather broad in application or that require the use of a GUI. With SkiRaff, we aim to help testers in writing less code for a single test. This will lead to less time spent on each test, enabling a wider test coverage. Covering more of the software will increase its overall quality. We also need the tests to be executed at a reasonable speed. If the tests are quick to write, but slow to execute, users may use other methods However, we deem this less important to how long it takes to write the tests.

The rest of this paper is organized as follows. Section 2 will give a brief overview of some related work. This is followed by an introduction to the pygrametl python package in Section 3. Section 4 will explain some basic testing terminology relevant to the paper. This leads into an overview of SkiRaff in Section 5. Section 6 shows how the framework performs setup before testing. In Section 7 we describe the classes used to represent the DW during testing. Afterwards, Section 8 explains the different types of predicates, used to assert properties of a DW. Section 9 will focus on evaluating the framework. Finally, in Section 10 we conclude upon the paper.

2. Related Work

In this section, we give a brief overview of the literature related to the work discussed in this paper. A lot of the literature discusses what methodology to use when testing ETL systems. While other articles focus on the development of frameworks to assist in testing. Both types of work are presented below.

ElGamal, et al., [4] analyses current methodologies in the testing of DWs. It is found that no approach provides full coverage of the different components of a DW. The ETL system is considered one of these components. A methodology is presented that emphasizes the use of automation during the testing of components.

Iyer [5] describes a real life example of automating ETL testing in a team of developers. By developing their own test framework, testers are able to perform testing earlier in the development cycle and with less effort. This leads to a better quality product and allows for development to be more agile. However, they focused more on methodology, and they did not discuss their framework in detail. Adding on to this Manjunath, et al., [6] empirically evaluates the benefits of automating regression testing using Informatica. Compared to manual testing, they find that there is a reduction in test cycle time by 50-60% and a reduction in effort cost by 84%.

Dakrory, et al., [7] presents a framework for automatic data centric testing of ETL systems. As opposed to SkiRaff, it generates its own test cases. Yet testing is not performed upon an implemented ETL as in SkiRaff. Instead it uses the ETL design documents in place of the implementation.

In [8] Thomsen and Pedersen describes the ETLDiff framework. Given two ETL systems, the framework is able to compare them, and note differences in how they process data from sources to DW. This is used for regression testing to ensure that regression does not occur during further development of an ETL system. Unlike SkiRaff the framework is not able to test for new features only regression.

The papers mentioned above indicate that automated tests have a central place in the field of ETL testing. There is a consensus that its usage can be very beneficial to the development of ETL systems. The literature does not cover in-depth any framework, which can be used to test new features of an implemented ETL system. As this is one of the features of SkiRaff, this work seems to hit a niché in the literature.

3. pygrametl

In this section, we give a quick outline of the python package pygrametl, as SkiRaff is developed to work in conjunction with this framework. The section is based upon the pygrametl documentation and its source code[9].

The package was developed at Aalborg University in Denmark and was released as open source in 2009. It has since then found use in a variety of different systems. It is developed to let users develop ETL programs by coding in python. This stands in contrast to other products on the market, which mainly allow their ETL development through a GUI. By allowing expert developers to use an API instead, it is believed that they will be more productive.

pygrametl provides assistance for the implementation of all three parts of an ETL system. Classes in the datasources.py module allow for the extraction of data from sources. A source can either be a CSV file or an SQL database. Data is always extracted as dictionaries. Each dictionary corresponds to a row from the source, and each key-value pair matches an attribute-value pair of that row. Transformation of the extracted data is supported by the classes found in steps.py. Although users often perform their transformations through normal python. Afterwards, loading is performed using the connectionwrapper class found in init.py and the classes found in tables.py. A connectionwrapper object is used as the default connection to the DW that must be populated. The table.py classes support the use of different types of dimension and fact tables. Once a table class is instantiated it is associated with an actual table in the DW. It can then be used to load data into that table.

Note that pygrametl never sets up sources or DW, it merely provides a way to work with them at an abstract level. They need to be set up separately by developers, and pygrametl must be used in conjunction with a python module that can access them. Such modules are often developed to be used with a specific Database management systems (DBMS). However, pygrametl only works with modules that conform to the PEP 249 standard [10]. This standard encourages similarity between modules that access databases. Because of this, pygrametl works with a broad range of DBMS'. To access databases, pygrametl generates SQL code dynamically. The

SQL generated is not DBMS specific and uses only standard keywords such as UPDATE, SELECT and JOIN. This is done to ensure compability with the different PEP 249 modules and their related DBMS'. SkiRaff aims for the same compatibility and avoids more DBMS specific syntax and functionality when generating SQL code. This also makes accessing metadata on a database or table difficult, as PEP 249 does not enforce a standard way to do this.

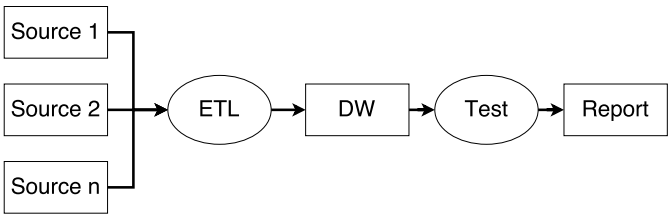

Figure 1. An example of a source to target test used for ETL

4. Software Testing

This section introduces the software testing terminology, that is important in understanding the content of later sections. The first subsection describes the general terminology, while the second goes more into the specifics of ETL testing.

4.1 Basic Testing

In this subsection, we describe some core concepts relating to software testing. The information presented here is taken from the books *Guide to Advanced Software Testing* by Hass[11] and *Software Testing Techniques* by Beizer[12].

Testing is an integral part of the software development process. It is included in many different software process models. Hass defines testing as follows:

> *Testing gathers information about the quality of the object under test.*

Knowing the quality of a piece of software allows an organization to evaluate, whether it upholds the requirements they set for it. If not, test results may guide an effort to increase quality.

Before tests can be executed, they must be designed. Design can be done from two different points of view: *Functional* and *Structural*. With functional, software is treated like a black box receiving input and computing output. Tests are designed to verify, whether the software conforms to specified behaviour. Structural involves designing tests based on the internal structure of the software. A tester could for instance test how a piece of code logically branches during execution. The structural approach is often used, when testing individual components. Functional is likely to be used, when testing a set of integrated units or an entire system.

During the software testing process, a system is usually tested at different structural levels. During *component testing*, each component of the system is tested in isolation. This usually requires the implementation of stubs and drivers. A stub simulates the code called by the component, while the driver simulates code, which the component is called by. The size of a component is defined by the organization, in which the code is tested. However, we can usually define a component as an aggregation of units. A Unit is the smallest testable part of a system, and it is itself defined to be a component. The testing of a single unit is often referred to as *unit testing*. The difference between unit and component testing is that component testing considers not only units but also aggregated components. Through component testing we verify the quality of each component. Once verified we begin to integrate the components with each other. Here *integration testing* may be performed. This type of test verifies the quality of the communication between the different components. After the system has been fully integrated, testers can perform *system testing*. At this level of testing, we test the system as a whole. Thorough testing usually involves testing the software at each structural level in the order described. Structural testing is usually performed at the component and early integration level. Functional testing is used for both the later integration and system level.

As with any other process, testing is restricted by both time and resources available. These greatly affect the quality of testing performed, often called *test coverage*. Because of this, testers are almost never able to perform exhaustive testing, where all combinations of input and preconditions are tested. Instead, testers select the most important areas to focus on during testing.

Tools are often used to assist in the testing process. An example is the testing framework, which often assist in the automatic execution of tests. A framework can execute a set of user-defined test cases, setting up and tearing down objects as needed. A test case is a well defined set of preconditions and input(s) given to a component. This is accompanied by a set of expected output(s) and postconditions. It either passes or fails, depending on whether the actual output and postconditions match expectations. The required test coverage is often acquired by designing a set of test cases. An advantage of frameworks is that testers can spend more time on test design. Yet, less time will be spent on how to implement tests. This leads to testing of a higher quality.

Another bonus of using a framework is that it allows for *regression testing*. This becomes mostly relevant in the maintenance part of the software life cycle. During this time, new features may be added and bugs fixed. Testers want to ensure that such changes do not lead to regression, where software quality deteriorates. With a framework, regression testing is simple. Once a set of test cases have been set up, they can be run after a change has been made. This will help in gauging whether a regression has occurred.

4.2 ETL Testing

According to Rainardi in his book *Building a Data Warehouse: With Examples in SQL Server*[13] ETL testing assures that changes to sources are captured and propagated to the DW properly. Therefore, we can describe the core of ETL testing

as being data-centric. Performance testing, which focuses on the speed of the transfer, is also relevant. However, for SkiRaff we choose to focus on ETL testing as more of a data verification activity. Assertions are often used when testing an ETL system. They allow testers to assert about the program state. It can then be reported on whether the assertion held. As an example, a tester can assert that a variable has the value 5. Testers may make assertions at the component and early integration level. At these levels on-market unit testing tools may be used. As users of pygrametl are thought to be experts, we assert that they know how to perform this type of test. As such, developing a framework to assist in this process is not necessary.

Instead we choose to focus on testing at the system level, which is often called *source to target* testing. An example is illustrated in Figure 1. Here the ETL program under test populates a DW using a set of sources. Once the process has been finished, we test and report on whether assertions about the resulting DW have been met. Note that these assertions are different from those discussed earlier, as they function on a higher level of abstraction. They concern properties of the DW and its tables such as table length, functional dependency etc. This type of test is functional, as it does not test according to the internal structure of the system. Testing is performed according to system requirements. This method is also ideal for regression testing, as old assertions about DWs can simply be checked again.

What should SkiRaff test during a source-to-target test? According to Rainardi [13], ETL testing is about ensuring that the data from sources ends up at the right locations of the DW. Thus data loss is a major concern. The checking of business rules is declared as a separate area of test. However, we argue that business rules are an important part of ETL testing. An ETL process should only load data into a DW, which conforms to agreed upon rules. Therefore we define the focal point of ETL testing to be both data loss and business rules. Both are touched upon in the remainder of this section.

4.2.1 Data Loss

Through an ETL process, data from sources are propagated into a DW. This new data contains information relevant to future business analysis. Yet, information may get lost during execution. Data loss can occur in any of the three sub parts of the ETL process. The wrong data may be extracted from the sources. Transformations may be faulty and produce incorrect results. Loss during load may happen through truncation. Truncation occurs, when the data type of a DW attribute can not store the amount of data necessary. This often results in data being cut to fit the data type. Any type of data loss results in a DW that does not contain required information. This may lead to faulty business analysis.

4.2.2 Business Rules

The term business rule is generally applied to any rule that states some constraint on the data stored in a DW. These may be unique to the business in question. For an airline, a rule could be that any passenger with over 10 flights in the last year should be registered to receive a special discount on snacks. Rules could also indicate attribute relationships such as hierarchies. These may state that a city must be located in a country.

Some more general business rules can be used to assert data integrity. Enforcing data integrity ensures the quality of data in a DW. Data integrity can be enforced using the following three types of rules:

- Entity integrity: Every table has a primary key, which is unique and not null for each row
- Referential integrity: A foreign key has to refer to an existing primary key or be null
- Domain integrity: An attribute should stay within its defined range of values

Data integrity is important, as it allows us to assert certain truths about our database. For instance, we know that each tuple is identifiable. Despite these advantages, DBMS' may not enforce them during loads. Loading data into a DW using an ETL usually involves a large amount of data, and access to the DW may be shut down temporarily. Enforcing data integrity during load is expensive. Therefore, a fast load often takes precedence over a safe load. If an ETL system is improperly designed, the DW ends up not having data integrity, which hinders its usefulness.

5. Overview of SkiRaff

In this section, we give an overview of SkiRaff and its major components. The sections following this go into detail about some of the individual parts of the framework. SkiRaff can be downloaded at our Git repository[14].

The purpose of SkiRaff is to assist in functional source-to-target testing of ETL programs developed using pygrametl. As such, testing is performed by asserting about properties of DWs, which have been populated by a pygrametl program. The focus of testing lies on data loss and business rules. Testers can make assertions about new functionality or perform regression testing using old assertions. Testing is performed at the system level. Thus, the extraction, transformation and loading components are integrated prior to test. The framework is not designed for exhaustive testing. It is instead meant to assist in reaching an adequate level of test coverage.

SkiRaff contains four major components:

- DWPopulator
- DWRepresentation
- Predicates
- Case

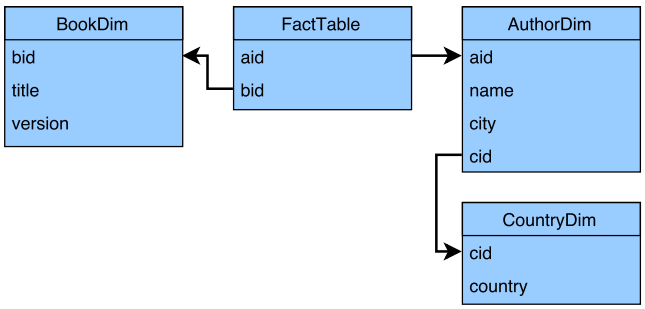

Figure 2. BookAndAuthor DW schema

The purpose of the DWPopulator is to execute the pygrametl program under test and populate the corresponding DW. When the DW is populated, we can run tests on it. Testers may choose, how the program is executed. They may just run it as usual, using the connections to sources and DW hardcoded into the program. However, these data sources may contain actual data collected and used by an organization. Because of regulations, such data may not be available for testers to use. Instead, SkiRaff allows testers to dynamically insert test sources and DW into the program before execution. This way, test data may easily be switched in and out without messing with the program itself.

After executing the program, the DWPopulator builds a DWRepresentation object corresponding to the populated DW. DWRepresentation is a class made to contain structural information about a DW. It also allows for easy access of the DW tables and their metadata.

The Predicates are a family of classes used to make assertions about the properties of a populated DW. All of them inherit from the Predicate class. Testers assert by instantiating a predicate class with the tables of interest, along with some additional arguments depending on the predicate type. Once instantiated, a predicate can be executed to check, whether the defined assertion holds or not. After execution, it creates a Report object that indicates the result of the check.

Case is a class representing a test case. It serves as the tester's primary way of interacting with SkiRaff. The class is rather simple, executing a set of predicates on a DW through a DWRepresentation. After a predicate is executed, Case displays the contents of the Report object made by the predicate to the user. Note that since Case takes a DWRepresentation as input, the user does not have to run DWPopulator to perform testing. Testers may construct a DWRepresentation themselves.

5.1 Example Scenario

This subsection will be used to describe an example scenario, in which SkiRaff is used to conduct tests. For this example, we imagine that we have already written a pygrametl program. This pygrametl program performs the necessary ETL operations to populate the BookAndAuthor test DW using two databases as sources. All three are constructed with SQLite, using the python module sqlite3, which upholds the PEP 249 standard. But any other module supporting PEP 249 could have been used. The DW schema of BookAndAuthor can be seen in Figure 2. E.g. we wish to test that our program correctly populates the DW, so that the fact table has exactly 99 rows. We also assert that there is referential integrity. This test is performed through the python script in Code Example 1. We explain the code below, with an in-depth description of each module used in later sections.

```
1  import sqlite3
2
3  # We execute a pygrametl program using our specified
4  # sources and create a DWRepresentation.
5  program = './bookandauthor_program.py'
6  src1 = sqlite3.connect(datasource='./db1')
7  src2 = sqlite3.connect(datasource='./db2')
8  dwp = DWPopulator(program=program,
9                    program_is_path=True,
10                   pep249_module=sqlite3,
11                   datasource='./dw'
12                   replace=True,
13                   sources=[src1, src2])
14
15 dwrep = dwp.run()
16
17 # We create our predicates
18 p2 = RowCountPredicate(table_name='FactTable',
19                        number_of_rows=99)
20 p1 = ReferentialIntegretyPredicate()
21
22 # We create our Case and run it
23 c = Case(dw_rep=dwrep,
24          pred_list=[p1, p2])
25 c.run()
```

Code Example 1. Example of a SkiRaff program

line 3-15 As we are using both test sources and a test DW, we wish to have these used as replacements for the data connections, which are hardcoded into the pygrametl program. This requires that the tester first creates PEP 249 connections to the sources. We then instantiate the DWPopulator with some parameters. These describe the program to run, how to connect to the test DW as well as the replacement sources. After instantiation, we execute the object by calling the `run` method. This runs the pygrametl program with the test sources and DW. It then returns a DWRepresentation object corresponding to the BookAndAuthor test DW.

line 16-20 We instantiate two Predicate objects. One asserts that the fact table has exactly 99 rows. The other asserts that there is referential integrity in the DW.

line 21-25 We instantiate a Case with our DWRepresentation and Predicate objects. We then call its `run` method. This runs each predicate on the populated BookAndAuthor DW. When each predicate has been executed, their results are reported to the tester.

In the following sections, we go into detail about the implementation of the three major components of SkiRaff, as discussed in this section. These include DWPopulator, DWRepresentation as well as all of the Predicate classes. Case will not be covered as its implementation is trivial.

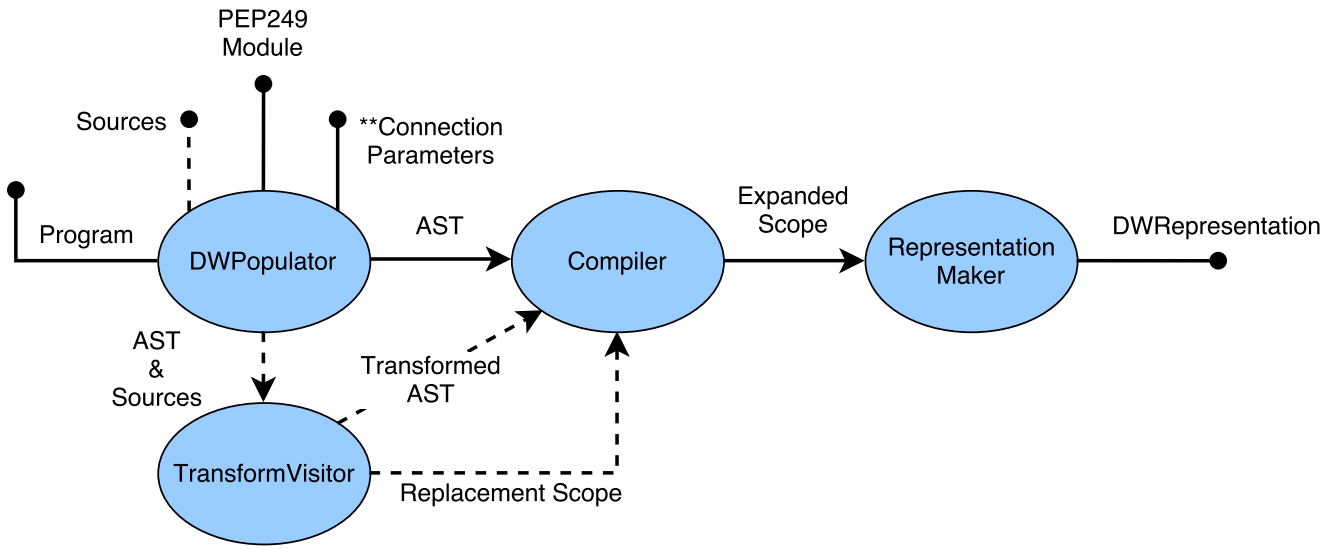

Figure 3. Data flow diagram of the DWPopulator. The dotted lines are an alternative path, used of we decide to replace sources and DW in the program.

6. DWPopulator

In this section, we describe the implementation of the DWPopulator component. Its is used to populate a DW using a pygrametl test program. After program execution, it creates and returns a DWRepresentation object. This object is used by the predicates to gain access to DW tables and metadata. On Figure 3 is a dataflow diagram, which illustrates the flow of data through the DWPopulator. Please note that this depiction is somewhat abstracted from the actual implementation to ease explanation. In the following subsections, we describe each of the four sub-components shown on the figure. Following this, we describe what restrictions the component operates according to. In this section we refer to connection objects. These are used in python to connect to data containers. In this context a connection object may either refer to a file handle, when connecting to a CSV file or a PEP 249 connection, when connecting to an SQL database,

6.1 TransformVisitor

Before execution of the pygrametl test program, testers may choose to dynamically replace the program's hardcoded sources and DW. If they opt to do this, the TransformVisitor class is used. Otherwise this component is skipped. The TransformVisitor takes as input the abstract syntax tree (AST) of the pygrametl program being tested, along with the connection objects for the test sources and DW, which we wish to use as replacements. Note that we always connect to a DW with a PEP 249 connection. The component walks over the AST of the pygrametl program, replacing the program's connection objects with those given by the tester. It inherits from ast.NodeVisitor and overwrites the visit_Call method. This method is called, when a call node implementing a call to a method or function is encountered. The overwritten method is shown in Code Example 2.

```
1  def visit_Call(self, node):
2      """ The visit of a call node.
3       Is an overwrite of Visit_Call ignoring all calls
4           except for those we
5       need to modify.
6      :param node: A call node
7      """
8      name = self.__find_call_name(node)
9      if name in ATOMIC_SOURCES:
10         id = self.__get_id()
11         self.__replace_connection(id, node)
12
13     elif name in WRAPPERS:
14         if self.dw_flag:
15             raise Exception('There is more than one
16                 wrapper in this program')
17         else:
18             id = self.dw_id
19             self.__replace_connection(id, node)
20             self.dw_flag = True
```

Code Example 2. The visit_Call method of TransformVisitor

The method only reacts to certain pygrametl method calls. After extracting the name of the method being called, we react if the name is contained in either `ATOMIC_SOURCES` or `WRAPPERS`.

`ATOMIC_SOURCES` is a list of the non-aggregate sources SQLSource, CSVSource and TypedCVSSource. All other sources from pygrametl.sources are aggregates of these three. Thus there is no reason to react to these. If the method name is contained within `ATOMIC_SOURCES`, it means that a non-aggregate source is being instantiated. One of the parameters used for such an instantiation is a connection object. By replacing this with the connection object of a test source, we force the non-aggregated source to point at the test data. This means that the object can now be used to access test data, rather than the data it was hardcoded to do. The replacement process itself consists of replacing the hardcoded connection parameter with a dummy key from the replacement scope. For the first source encountered the key will be `__1__`, the second

__2__ and so on. These will later be used as identifiers, when executing the AST.

WRAPPERS contains only ConnectionWrapper from pygrametl.init. This is the class used to access DWs with pygrametl. Once the instantiation of a ConnectionWrapper is encountered, we simply replace its connection parameter with the key __0__. We also set a flag that makes sure that we raise an exception if another ConnectionWrapper is encountered. We do this to restrict SkiRaff to only work with pygrametl programs that function on a single DW.

Once the entire tree has been walked, a transformed AST has been produced. We then create a replacement scope, __0__ points to the DW connection and __1__, __2__, etc, points to their corresponding source connections. The transformed AST and the replacement scope is then sent to compilation.

6.2 Compiler

In this component, we compile and execute the pygrametl program. If we used the TransformVisitor, we compile the transformed AST and execute it using the replacement scope as local scope. Otherwise, we simply compile the regular AST, and execute it with an empty local scope. In both cases, this is done with calls to the built-in python functions; compile and exec. In python, a scope is given by a dictionary, where keys are variable names paired with object references. In the case that we used TransformVisitor, we replaced the connecting objects in the transformed AST with keys from our own scope. By executing using the replacement scope, we force the program to use the test objects found in that scope. Thus we overwrite the hardcoded sources and DW from the program. Once the AST has been executed in either case, the test DW will be populated, and whatever local scope we used will have been expanded with the variables found in the executed pygrametl program. The expanded scope is sent to the RepresentationMaker. After execution we also make sure to reestablish connection to the test DW, in case it was closed during execution.

6.3 RepresentationMaker

Using the expanded scope, the RepresentationMaker accesses all Dimension and FactTable objects instantiated through the execution of the pygrametl program. From this, it creates a DWRepresentation object that allows for easy access to all the tables of the DW along with their metadata. It begins by instantiating the RepresentationMaker, then calling its run method to create the DWRepresentation object. This method can be seen in Code Example 3

In this method we access the _alltables list from our scope. Every pygrametl Dimension and FactTable object instantiated during execution of the pygrametl program is logged in this list. Once acquired, we iterate over the list and create a representation object for each table. check_table_type is a method that returns true, if the table in question has a type contained in a given list. DIM_CLASSES contains the class names of all non-aggregate Dimension classes from pygrametl.tables. Similarly, FT_CLASSES contains

```
def run(self):
        pygrametl = self.scope['pygrametl']
        tables = pygrametl._alltables

        # Creates representation objects
        for table in tables:

            # If the table is a dimension.
            if self.check_table_type(table, DIM_CLASSES):
                if isinstance(table,
                     TypeOneSlowlyChangingDimension):
                    dim = SCDType1DimRepresentation(table
                          , self.dw_conn)
                elif isinstance(table,
                     SlowlyChangingDimension):
                    dim = SCDType2DimRepresentation(table
                          , self.dw_conn)
                else:
                    dim = DimRepresentation(table, self.
                         dw_conn)
                self.dim_reps.append(dim)

            # If the table is a fact table
            elif self.check_table_type(table, FT_CLASSES)
                 :
                    ft = FTRepresentation(table, self.
                         dw_conn)
                    self.fts_reps.append(ft)

        #SnowflakedDimensions
        snowflakes = []
        for x, value in self.scope.items():
                if isinstance(value, SnowflakedDimension)
                     :
                    snowflakes.append(value)

        dw_rep = DWRepresentation(self.dim_reps, self.
             dw_conn, self.fts_reps, snowflakes)
        pygrametl._alltables.clear()

        return dw_rep
```

Code Example 3. The run method of the representation_maker class

all non-aggregate fact table classes with the exception of SubprocessFactTable. Once the type of a table has been identified relevant metadata such as table name and keys are extracted from the object and used to instantiate a DimensionRepresentation or FactTableRepresentation respectively and stored in a list. After iterating, we identify all SnowflakedDimension objects in our scope and store them in a list. These provide some assistance in discovering the structure of the DW. Afterwards, the DWRepresentation object is instantiated with the two lists of tables, the list of SnowflakedDimensions and the PEP 249 connection object connecting to the DW.

Once instantiated, the DWRepresentation is returned to be used by the predicates and the reinterpretation process is completed.

6.4 Restrictions

In this section, we describe the restrictions of the DWPopulator.

The dynamic alteration of code can become rather complex, if each edge case is considered. Thus, we decide to restrict the format of pygrametl programs that can be run through TransformVisitor. As mentioned previously, the program may only function on a single DW. We make this restriction as it makes implementation easier. Also the use of more

than one DW is uncommon, so applying this restriction will not be an issue for most users. As we replace connections by walking over the AST, pygrametl programs may not instantiate several source or table objects through iteration. As an AST walker only moves over the code within a loop a single time. Finally, the instantiation of pygrametl sources and tables may only occur in the script, from which the AST is created. Imports are still usable, but because the AST walker only traverses the given script, replacements will not be performed inside imported modules.

In the RepresentationMaker, we restrict ourselves from extracting metadata from instantiations of the SubprocessFactTable class. This class is used to write data to some sub processes such as a bulkloader. We ignore it, as it does not contain the needed metadata such as a table name

All of these restrictions can be circumvented by not performing replacements of sources and DW.

7. Intermediate Data Representations

This section will go into details about the different intermediate data representations used by the predicates. These representations are meant to standardize input to the predicates, so that the DW may be easily accessed. There are two main intermediate data representations *DWRepresentation* and *TableRepresentation*. DWRepresentation represents the DW as a whole, and the TableRepresentation and its subclasses represent the DWs tables. A UML diagram of the intermediate data representations can be seen in Figure 4. Both classes are involved in the execution of the DWPopulator class as described in Section 6. The metadata stored in both classes is used when dynamically creating SQL queries in the predicates. However both also contain methods that assist in extracting data, so that it may be processed using python.

7.1 DWRepresentation

The *DWRepresentation* is a container object and includes information about the DW's schema and contents. A DWRepresentation object is created by the *RepresentationMaker* by aggregating a number of *TableRepresentations*.

The DWRepresentation has the following main attributes:

Dimensions List of *DimRepresentations* detailing each dimension in the DW along with their metadata.

FactTables List of fact tables in the DW along with metadata.

Snowflakes List of Snowflaked dimensions. This is used to construct the DW schema.

Connection A PEP 249 connection object for accessing the DW, so that we may query it for data later on.

The object also provides methods for connecting to and extracting data from the DW. Data can be extracted in one of two ways.

By supplying the name of a table to the get_data_representation method, it will return the corresponding TableRepresentation object. DWRepresentaion also supplies the iter_join method. Giving this method a number of table names as input, it will return an iterable. This way we can extract and iterate over the natural join of the given tables as a list of dictionaries within python.

During instantiation, the DWRepresentation also finds the DW's schema . This is done using the _find_structure method. For each fact table, we register all dimensions, which it may be naturally joined with. That is, an attribute in the fact table's keyrefs set is the same as the primary key of a dimension. In the case of our snowflaked dimensions, a fact table may only have a foreign key to the snowflake root. Dimensions within snowflakes are the only ones allowed to have a foreign key to other dimensions. Doing this lookup for each fact table, we end up with a dictionary, where the keys are fact table names pointing to sets of dimensions. An example would be {ft:(A,B)}, which states that the fact table ft has foreign keys to the dimensions A and B. To this dictionary we add the internal reference information of each snowflaked dimension. Thus it also contains keys for dimensions with foreign keys to other dimensions within snowflakes. Because how we find the structure, pairs of referring and referred tables need to be joinable through a natural join.

7.2 TableRepresentation

TableRepresentation is a superclass used for representing data from specific tables. The subclasses contains metadata regarding the tables themselves, such as the table name, attributes etc. A subclass is instantiated by taking an instance of its corresponding pygrametl table as input. It then copies the relevant metadata and PEP 249 connection object into itself.

The currently supported table types are:

- Dimension
- Type 1 Slowly Changing Dimensions
- Type 2 Slowly Changing Dimensions
- FactTable

Any dimension or fact table found in a pygrametl program is simply represented in a DimensionRepresentation or FactTableRepresentation respectively. Special subclasses are made for the slowly changing dimensions, as they contain unique information important during the execution of certain predicates.

When working with DW data in python, rather than SQL, the data within a specific table can be extracted by iterating over the TableRepresentation as a list of dictionaries. This queries the DW and yields a single dictionary corresponding to a row at a time. It also has the itercolumns method, which allows for only a specified subset of table columns to be iterated in the same manner.

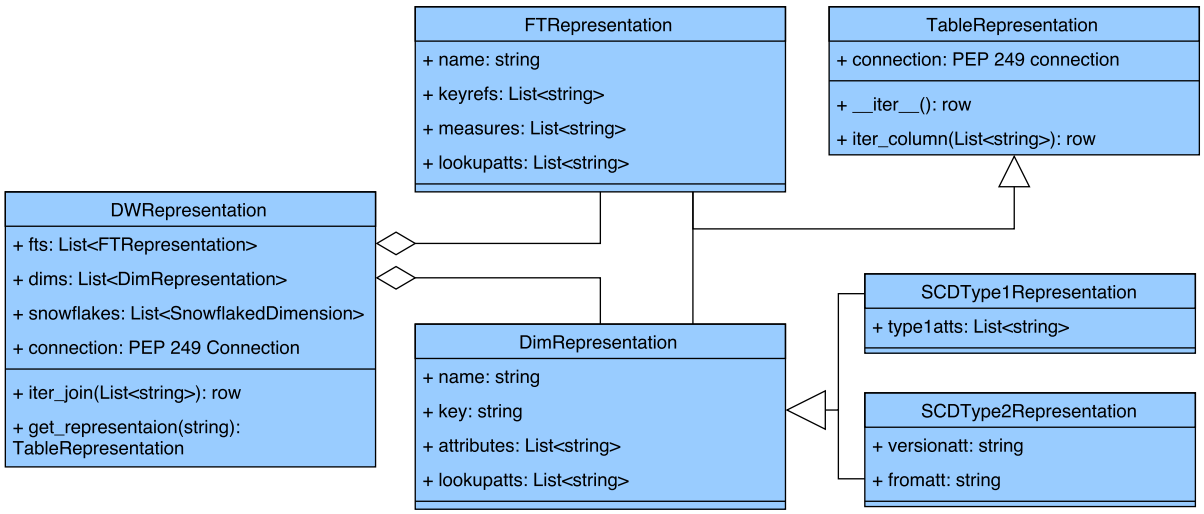

Figure 4. UML diagram of the intermediate representations

8. Predicates

This section will describe each of the predicate classes found in SkiRaff. These are used to assert specific properties of a DW during test. Each subsection will describe the purpose of a predicate, how it is instantiated and how it functions. Each predicate is related to Figure 2 as to show, how it may be used in practice. In the implementations we have tried to use as much SQL as possible, where it made sense to do so. Working with data close to the database is much faster than treating representation objects within Python. One complication with this approach is that different DBMS' usable with SkiRaff use different implementations of SQL. Thus we risk damaging the compatibility of SkiRaff by using SQL. With this in mind, we restrict the predicates to use only the most standard SQL functionality. The SQL queries in question are dynamically generated by the predicates, depending on how they are instantiated. Across the predicates these queries often return all rows found to be faulty according to the assertion. If no rows are returned, a successful assertion is reported. If rows are returned, a failed assertion is reported along with the faulty rows. In general, more in-depth reporting upon predicates only occurs, when they fail.

For instantiation of different predicate types some common parameters are used. These are presented here, as not to repeat the information later.

table_name a string with the name of a table from the DW. Using the get_data_representation method from DWRepresentation, this is used to fetch the corresponding table. Can also be given as a list of strings, if a natural join of tables is wanted insead. Most of the predicates may function on a join of tables. If they do not, we specify it in the individual sections.

column_names a string or list indicating one or more attributes from the table. If the attribute is None, we fetch all attributes for each row. If not, the data fetched depends on column_names_exclude.

column_names_exclude a boolean that indicates, which attributes to work on. If false, we use the attributes as described in column_names. If true we use all attributes not in column_names. Default is false.

8.1 RowCountPredicate

RowCountPredicate asserts if a table has a specified number of rows. This can be useful in situations where the user knows how many rows should be in a given table after load. RowCountPredicate is instantiated as follows, with description of the parameters in Code Example 4.

```
1 RowCountPredicate(table_name='CountryDim',
2                   number_of_rows=200)
```

Code Example 4. Instantiation of RowCountPredicate

number_of_rows an integer representing the specified number of rows that the table should have

In the instantiation shown in Code Example 4, we assert that CountryDim has 200 entries. RowCountPredicate would in this instance generate the following SQL query, which counts how many rows are in a given table.

```
1 SELECT COUNT(*)
2 FROM CountryDim
```

Code Example 5. SQL query generated from Code Example 4

RowCountPredicate will then compare the scalar result from the query in Code Example 5 to the specified number of rows we are looking for, in this case 200. If equal, the test is successful, if not equal, the test return false and reports how many rows are actually in the table.

8.2 ColumnNotNullPredicate

For a set of specified columns within a table, ColumnNotNullPredicate lets testers assert that no member of one or more columns are null. While nulls may be allowed to occur within a table, they could indicate data loss. It may also be the case that not null constraints are placed upon certain columns. This is the case for primary keys. The ColumnNotNullPredicate is instantiated as exemplified in Code Example 6

```
ColumnNotNullPredicate(table_name='BookDim',
                       column_names=['bid'],
                       column_names_exclude=False)
```

Code Example 6. Instantiation of ColumnNotNullPredicate

For the instantiation, we assert that no row in the column bid in the BookDim is null. This is important to check, as bid is a primary key to the dimension. In this case ColumnNotNullPredicate generates the query shown in Code Example 7

```
SELECT *
FROM Bookdim
WHERE bid IS NULL
```

Code Example 7. SQL query generated from Code Example 6

This query fetches all rows that contains null in the column bid. If any rows are found, the assertion fails and all found rows are reported to the user. Otherwise it succeeds.

8.3 NoDuplicateRowPredicate

NoDuplicateRowPredicate asserts that a given table contains only unique entries. This is often important, when we want to ensure no duplicates of primary keys. Using the predicate, the tester may also indicate, what columns should be considered as part of the assertion. As such, we may specify that a whole row does not have to be unique, only a subset of its columns. This is useful when asserting unique keys or lookup attributes. In Code Example 8 is an example of how the predicate can be instantiated.

```
NoDuplicateRowPredicate(table_name='AuthorDim',
                        column_names=
                            ['aid'],
                        column_names_exclude=False,
                        )
```

Code Example 8. Instantiation of NoDuplicateRowPredicate

In the instantiation displayed, the predicate asserts that each entry in the aid column of AuthorDim is unique. This is important as aid is the primary key to the table. NoDuplicateRowPredicate will in this case generate the SQL query shown in Code Example 9, to satisfy the previous assertion.

This query groups together all rows on aid, and fetches groups with more than one member in AuthorDim. These groups are duplicates. If any are found the assertion fails, and the duplicates are reported to the tester.

```
SELECT aid, name, year, COUNT(*)
FROM AuthorDim
GROUP BY aid
HAVING COUNT(*) > 1
```

Code Example 9. SQL query generated from Code Example 8

8.4 ReferentialIntegrityPredicate

ReferentialIntegrityPredicate asserts that there is referential integrity between tables in a DW. This means that each row of a table with a foreign key, has a corresponding row in the table to which it is referring. Referential integrity is not always upheld during load, so a tester may want to assert this property. The ReferentialIntegrityPredicate can be instantiated like shown in Code Example 10, followed by an explanation of the parameters.

```
ReferentialIntegrityPredicate(
    refs={'FactTable': ('BookDim', 'AuthorDim'),
          'AuthorDim': ('CountryDim')},
    points_to_all=True,
    all_pointed_to=True
)
```

Code Example 10. Instantiation of ReferentialIntegrityPredicate

refs a dictionary containing names of tables. It has the same structure as in Section 7, pairing a table with the tables that it has foreign keys to. The dictionary indicates, which references we want to check integrity for. If the programmer wants to run the predicate on the whole DW, it is redundant to provide this parameter, as the DWRepresentation object already holds information about the DW schema. Thus the default is None.

points_to_all A boolean which is True by default. If it is set to True, the predicate checks if referring tables have corresponding foreign keys in the tables they refer to.

all_pointed_to A boolean which is True by default. If it is set to True, the predicate checks if referred tables have corresponding foreign keys in the tables they are referred by.

Either of the two booleans can be set to false, but not both. If both are false, it is the equivalent of having the predicate check references for no tables, and so an error will be raised instead. If the programmer wishes to check references for all tables in the DW, he can simply not provide any parameters, and let them use their default values.

In Code Example 11 is one of the queries generated in the case of the instantiation. It returns all rows from FactTable, which refers to a non-existent row in AuthorDim. Queries like this are generated for each one-way check for references between two tables, i.e. for `points_to_all` and

all_pointed_to. If any of the generated queries returns rows, the assertion fails and relevant information for each reference is reported to the tester. This includes: the key name, the key's value, the table that had a reference for it and the table that did not.

```
SELECT *
FROM facttable
WHERE NOT EXISTS(
    SELECT NULL FROM author_dim
    WHERE facttable.aid = author_dim.aid
    )
```

Code Example 11. One of the SQL queries generated from Code Example 10

8.5 FunctionalDependencyPredicate

FunctionalDependencyPredicate asserts that a table holds a certain functional dependency. This can prove useful to check if a DW holds certain hierarchical properties, i.a.. FunctionalDependencyPredicate is instantiated as follows, with a description of its parameters shown in Code Example 12.

```
FunctionalDependencyPredicate(table_name=['CountryDim',
                                          'AuthorDim'],
                              alpha='city',
                              beta='country')
```

Code Example 12. Instantiation of FunctionalDependencyPredicate

alpha attributes being the alpha of the given functional dependency. Given as either a single attribute name, or a tuple of attribute names.

beta attributes which are functionally dependent on alpha. Given as either a single attribute name, or a tuple of attribute names. For example with alpha as `('a', 'b')` and beta as `'c'` we get the functional dependency: $a, b \rightarrow c$.

In our instantiation above, we use the predicate to assert that there is a functional dependency between a book's title and its author. If this assertion holds, it means that no book is written by more than one author.

In Code Example 13 we see the query generated in this case. Here we join the table with itself on alpha, then return all distinct rows in which alpha can not be used to determine beta. If the query returns any rows, the assertion fails and the rows are reported to the tester.

```
SELECT DISTINCT t1.country, t1.city
FROM countrydim NATURAL JOIN authordim AS t1, countrydim
     NATURAL JOIN authordim AS t2
WHERE t1.city = t2.city
AND t1.country <> t2.country
```

Code Example 13. SQL query generated from Code Example 12

8.6 SCDVersionPredicate

This predicate allows testers to check if a given entry in a table has an asserted largest version. This predicate only works on SCDType2Representation objects, which represent type 2 slowly changing dimensions. SCDVersionPredicate is instantiated with parameters as depicted in Code Example 14.

```
SCDVersionPredicate(table_name='BookDim',
                    entry={'Title': 'The Hobbit'},
                    version=10)
```

Code Example 14. Instantiation of SCDVersionPredicate

entry A dictionary that pairs lookup-attribute names with values. A set of lookup-attributes being a key on the table, it is used to fetch all instances of unique row under assertion.

version The asserted largest version value of the test entry.

In the instantiation, we use the predicate to assert that The Hobbit's highest version number in BookDim is 10.

The predicate works through the query shown in Code Example 15. Here it uses *entry* to fetch all instances of the entry in *table_name*. Each instance having a different version number. The query then extracts the maximum value of the version attribute. The name of version-attribute is fetched from the metadata in the table's corresponding DimRepresentation object. In this case the name is simply 'version'.

```
SELECT MAX(version)
FROM BookDim
WHERE Title = 'The Hobbit'
```

Code Example 15. SQL query generated from Code Example 14

After having fetched the actual maximum value, it is compared to the one asserted by the tester. If the two values are equal, the assertion holds. Otherwise it fails.

8.7 CompareTablePredicate

CompareTablePredicate is by far the most complex of the predicates. It is used to assert that two tables uphold some type of comparison. The predicate allows testers to compare an actual table within the DW to one expected by the user. Users have different options, in how they would like to compare two tables. They may want to check for a one to one equality between tables, or whether the contents of the expected table is included in the other. In some cases, testers may want their comparisons to treat tables as sets, while at other times duplicates should be accounted for. Contrary to the other predicates, this does not only perform checks using SQL queries, but also using regular python code. This introduces a few unique challenges as covered in this section. The CompareTablePredicate is instantiated as follows, with a description of the parameters shown in Code Example 16.

```
CompareTablePredicate(actual_name='BookDim',
                      expected_table=[{'title': 'The␣
                          Hobbit', 'Version': 2}],
                      column_names=['bid'],
                      column_names_exclude=True,
                      sort=False,
                      sort_keys=(),
                      distinct=False,
                      subset=True
                      )
```

Code Example 16. Instantiation of CompareTablePredicate

actual_table Is a string with the name of a table from the DW, which should be compared with *expected_table*. It may be given as a list of strings, if a natural join of tables is needed instead of a single table.

expected_table Is the user-given table to compare against. If the table resides in the DW, then it may be given by a string containing its name. By giving a list of strings, a tester may instead get a natural join of tables as before. Instead of strings, the parameter may also be given a PEP 249 cursor, from which data can be fetched, or a list of dictionaries.

sort A boolean defaulting to True. If false we do a sort compare, else a unsorted comparison is performed. Both types of comparison are explained below.

sort_keys A set of attribute names to sort both tables upon, when doing a sort compare. Defaults to an empty set.

distinct A boolean defaulting to true. If true tables should are treated as distinct, while false means that duplicates should be accounted for during comparison.

subset A boolean defaulting to false. If true the comparison will check whether all entries in expected appear in actual. If false, it also checks that all entries in actual appear in expected.

For the instantiation above, we assert that BookDim contains the 2nd version of the Hobbit. We ignore the bid attribute for the compare, as it is a surrogate key. We do not perform sort compare. We also indicate that the tables should not be viewed as distinct. This is important, as the row within expected should only appear once in actual. We want to make sure that no duplicates of it exists within the actual table.

When running the predicate, it performs comparisons differently based on settings chosen, and how the expected table is stored. In general, the fastest execution comes from having the expected table contained within the DW. As such, testers may want to insert larger test tables into the DW. If expected table is given as a list of dictionaries or a cursor, it will be slower as comparison cannot be made at the database level. When a PEP 249 cursor is given, we create a list of dictionaries from it at instantiation.

The types of comparison can be split into two, sorted and unsorted. These will be explained in the following.

8.7.1 Sorted Compare

Sorted comparison works by sorting the two tables and then comparing them one by one. It works the same no matter, how expected table is given. If the comparison should not account for duplicate rows, we simply use the distinct keyword, when querying from the DW. When expected table is a list of dictionaries, we remove duplicates using python.

In order to perform a sorted compare, a sort key is needed. This key should be unique for each row in both tables. If two equivalent tables are sorted on such a key, a positional comparison between entries from both tables, should find no non-equivalent pairs of rows.

The tester may give a sort key. If one such key is not given, the predicate will try to generate one. This entails creating the union of one candidate key from each sub-table that joined together makes the actual table. Each table has at least one such key. Dimensions have their primary key and set of lookup attributes. Fact tables have their set of keyrefs. We can only construct the sort key, if a candidate key for each table can be found within the set of chosen columns, which we compare upon.

If a sort key is present, we can perform a sorted comparison. For tables in DW this entails joining together all the sub-tables, then using ORDER BY to order the tables according to the sort key. If the expected table is a list of dicts, the built-in python function *sorted* is used.

In the case where the comparison should check for equivalence, the following occurs: After sort, both actual and expected are iterated simultaneously using python. At each iteration, we fetch the next row of both tables and compare them. If they do not match, we break out of the iteration. We may also break out, if one of the tables is emptied before the other. Such premature breaks result in a failed comparison. A successful comparison occurs, if both tables are emptied at the same time.This indicates that the tables are of equal length and contain the same rows.

The method is rather similar, when checking whether the expected table is a subset of the actual table. This time we iterate over just the actual table and compare its rows to those of expected table. We only fetch the next entry from the expected table, when there is a match. The iteration is broken once one of the tables is emptied. Only if the expected table was emptied, meaning that all its elements found matches, do we report success. In any other case, a failed comparison is reported.

In both cases, comparison occurs through python. When we fetch a row, we convert it to a dictionary. This changes all stored nulls to the python None-type. This is problematic as Null = Null equals unknown, while None = None equals True. To mimic an SQL comparison, we treat all rows containing Nones as faulty. When using sorted comparison to check for equivalence, we break out of iteration and fail, if any row is found to contain None. In the case of subset comparison, we only break prematurely if a None entry is found in one of the expected rows. This precaution is used in all parts of the

predicate, where comparison is made using python.

Compared to other predicates, sorted compare does not report on faulty rows. It reports fail, once a single fault is discovered. This is partially because a single faulty row, changes the sort order. This will result in a lot of rows being marked as faulty by positional comparison, even though the may have a corresponding row in the other table. Such a report would not be helpful to the tester, so this type of comparison simply reports success or fail.

Sorted compare is the fastest comparison to perform in this predicate. This is the case, as sorting with SQL generally has an average running time of O(n log(n)), where n is the size of the table. The comparison takes at most O(m), where m is the size of the smallest of the two tables being compared. If the tables are not equivalent, the running time will most likely be cut short, as the predicate reports at the first fault found. Thus unsorted compare is well to use, when the tester simply wants to know whether two tables are equivalent, not interested in faulty rows.

8.7.2 Unsorted Compare

Unsorted compare is performed, when a sort key is not available, or the tester wants a more in-depth report showing faulty rows. In this case, comparison differs based on whether expected table is in the DW or a list of dicts. In this section we use tables from Code Example 16 for our examples.

In both cases, we perform set operations between tables. When checking whether there is full equality between the tables, we assert that:

$$A \backslash E \cup E \backslash A \equiv \emptyset$$

Evaluates to true. This indicates that $A \equiv E$, E is the expected table and A is the actual table. If the tester wants to check if the rows of the expected table is contained within the actual table, it is assert that:

$$E \backslash A \equiv \emptyset$$

Should be true. This indicates that $E \subseteq A$.

If the expected table resides within the DW, and when not accounting for duplicates, we use the following SQL query to assert $Expected \backslash BookDim \equiv \emptyset$.

```
SELECT  *
FROM Expected
WHERE NOT EXISTS (
SELECT NULL
FROM BookDim
WHERE expected.title = BookDim.title AND
expected.version = BookDim.version
)
```

Code Example 17. SQL query if expected table is in DW and we treat tables as distinct

If the query returns no rows, we know that $Expected \subseteq BookDim$ is true. If we want to check for the full equivalence, we perform a similar query, checking whether all rows within the actual table reside within the expected table.

In the case, where we have to take account for duplicates within the tables, we have to perform another query. For $Expected \backslash BookDim \equiv \emptyset$, this is shown in Code Example 18.

```
SELECT *
FROM (SELECT title,version,COUNT(*)
          FROM Expected
          GROUP BY title,version
WHERE NOT EXISTS (
SELECT NULL
FROM (SELECT title,version,COUNT(*)
          FROM BookDim
          GROUP BY title,version
WHERE expected.title = BookDim.title AND
expected.version <= BookDim.version
)
```

Code Example 18. SQL query if expected table in DW and we account for duplicates

By grouping together similar rows within each table, we can do the same type of comparison as before. Along with each grouped row, we also account for the amount of duplicates using the COUNT aggregate function. Thus we also need to extend the conjunction in the WHERE-clause with another logical clause concerning amount of duplicates. When doing a subset compare, the amount of duplicates for each grouped row in the expected table should be less than or equal to those of the actual table. When checking for equality between tables, we want the duplicates of each grouped row to be equivalent between the tables. Expect for these changes, we use the queries in the same way, as when not accounting for duplicates.

In the case, where expected table is a list of dicts, the comparisons are similar to the ones above. Instead of SQL, we use python composite statements and the filterfalse method from the itertools package, to find all faulty rows during comparison. In this case we also fetch the actual table into a list of dicts, using an SQL query. When comparison should not take account of duplicates, we remove duplicates from actual table using the DISTINCT keyword in the query. At the same time, we use python to remove all duplicates from the expected table. As all comparison is performed with python, we make sure to handle nulls, as we did during sorted compare.

Unsorted comparison is much faster to run, when the expected table is in the DW. Doing comparisons in python, we have to fetch all the data at once, and we do not get the benefits of having a database, such as indexing. The benefit of doing unsorted compare instead of sorted, is that it reports the faulty rows. However, it does have a longer execution time than sorted compare. Especially when taking account of duplicates.

8.8 RuleRowPredicate

RuleRowPredicate is used to assert that every row of a table complies with some user-defined constraint. Thus, this predicate allows users a lot of flexibility in how they test business

rules for individual rows through python code. The RuleRowPredicate may be instantiated as shown in Code Example 19.

```
def no_autobios(name, title):
    return not name == title

RuleRowPredicate(table_name=['AuthorDim','FactTable','
    BookDim']
                 constraint_function=no_autobios,
                 column_names=['name', 'title'],
                 constraint_args=[],
                 column_names_exclude=False)
```

Code Example 19. Instantiation of RuleRowPredicate

constraint_function a python function that represents the user constraint. It must return a boolean, indicating whether a given row conforms to the constraint.

constraint_args a list of additional arguments given to the constraint_function.

In the instantiation Code Example 19, we use the no_autobios as our constraint function. The predicate as a whole asserts that no author in the DW has written a book with their own name as the title. Autobiographies are generally expected to be of poor quality, and we do not want them in our DW.

With column_names we define, which row attributes that constraint_function should get as input. As we iterate over each row, we call constraint_function. The function receives the defined row attributes as parameters along with those from constraint_args. If the function returns false for a row, the assertion did not hold and all faulty rows are reported to the tester. Should be noted that all parts of this check are done at python level. This means that we need to fetch all data from the DW at once. Thus this will be slower to use than other predicates.

8.9 RuleColumnPredicate

RuleColumnPredicate is similar to RuleRowPredicate in section 8.8, even taking the same parameters as input and being performed mainly through python code. The difference is that RuleColumnPredicate allows for the assertion of business rules that apply to sets of columns. It is instantiated as shown in Code Example 20

```
def no_forgotten_cities(city, old_city_name):
    return old_city_name not in city

RuleColumnPredicate(table_name='AuthorDim',
                    constraint_function=
                        no_forgotten_cities,
                    column_names=['city'],
                    constraint_args=['Constantinople'],
                    column_names_exclude=False)
```

Code Example 20. Instantiation of RuleColumnPredicate

In this case, we give no_forgotten_cities as the constraint function. The predicate asserts that no city in the AuthorDim table is named Constantinople.

During execution, each column specified by column_name is fetched from the database in its entirety. The columns are then supplied to the constraint_function along with the additional parameters in constraint_args. Note that in contrast to RuleRowPredicate, we only call the function a single time, since it takes entire columns as input. The assertion made by the predicate holds, based on whether constraint_function returns True or False.

9. Evaluation

In the following section, we evaluate SkiRaff. The aim of the evaluation is to argue that SkiRaff lets testers write less code, and execute tests at a reasonable speed. A reasonable execution speed should not differ much from speeds of other test methods. In the following, we describe our criteria and how an experiment was performed. Finally we present our results.

9.1 Criteria of Evaluation

As mentioned in Section 1, no test framework on market is developed specifically for pygrametl. So there is no obvious candidate to compare against. We may evaluate against more generic on-market products, such as QuerySurge or AnyDBTest. However because of time constraints, we opt to not use these tools, as they need to be learned. Instead we choose to evaluate against manual testing, which tests code through SQL queries. The evaluation will show the potential gain of migrating to SkiRaff from manual testing.

Now that we have determined what to evaluate SkiRaff against, we describe the criteria used for the evaluation.

Number of statements used The best solution uses the fewest statements to test an ETL system. In general, as the amount of statements increase, a program becomes more difficult to comprehend. As the size grows, more time will have to be spend on general development and debugging.

Execution time The best solution spends the shortest amount of time on executing tests. Any test involving databases will often involve large amounts of data. An efficient runtime mean the difference between a test running for a few hours to days.

Note that we evaluate based mostly upon the number of statements used. We accept an increase in runtime, if the program becomes easier to comprehend.

Another possible criteria could have been implementation time. However, we have already worked in-depth with SQL queries internal when developing the framework predicates. These queries are very similar to the ones needed for manual testing. Thus we expect implementation time for manual testing to be rather fast, as we are already familiar with these queries. As this may not be representative of the time needed for manual testing, we decide against using this criteria.

9.2 Experiment

Having now defined our criteria, we perform an experiment, to gather data for the evaluation of SkiRaff against manual testing.

For the experiment we created a small pygrametl program to populate the BookAndAuthor DW as presented in Section 5. The source code for the program can be seen in Appendix A. For sources, the program uses two SQL databases, containing data on books and authors, and a single csv-file storing pairs of city and country names. Once the sources have been set up, an empty DW is populated using the pygrametl program. In order to populate the DW with dirty data, we do not enforce any integrity constraints in the DW, as is common during an ETL load. At the same time, the pygrametl program does not handle dirty data extracted from the sources. This leads to faults, such as missing keys, which we discover during the tests. Once the DW is filled, we perform tests using both SkiRaff and manual. Manual testing is done using SQLite through the sqlite3 python module. In both instances, testing covers the following test cases. They are chosen, as to use every predicate that SkiRaff offers.

- No nulls appear in any column of authorDim
- There is functional dependency between cid and city in the join between authorDim and countrydim.
- Referential integrity is upheld through the entire DW
- All (name,cid) pairs within authorDim are unique
- BookDim contains 6 rows
- In bookDim, the highest version value of the book “EZ PZ ETL” is 4
- Contents of goodbooksdim is a subset of bookDim. Goodbooksdim contains two rows for the books “The Hobbit” and “The Divine Comedy”. The comparison should not take account of duplicates.

During experimentation we strive to make sure that both tests deliver the same reports on the assertions. This is important as SkiRaff focuses on reporting faulty data to the tester. This does slow down performance, compared to if we merely reported whether a predicate passed or failed. To make the two methods comparable, manual testing must live up to the same requirement of verbosity when reporting. This is also why the sqlite3 module is used, instead of a simple SQLite script, so that it may print to the prompt in the same way af SkiRaff

The script written for SkiRaff can be found in Appendix B, and the corresponding script for manual testing is found in Appendix C. For transparency’s sake the scripts include the code used to report on execution time.

9.3 Results

After testing with both methods, we measure the number of statements used and the execution time of each. We do not count imports, code used to reporting on execution time nor string assignments as statements. When counting statements for manual testing specifically, we count each clause in an SQL query as a separate statement. For a WHERE-clause, each logical statement is counted as a clause itself. Execution of the test scripts are performed, where AuthorDim has 9988 rows, BookDim 9996, CountryDim 2 and FactTable 10007. The results of the measurements can be seen in Table 1.

In Table 1 we can clearly see that SkiRaff allows testers to use less statements in their tests. This strongly indicates that more expressive power is contained within each statement. Looking at the execution time, we can see that the two approaches are almost identical in their execution time, both with setup and predicates respectively. ”Execution Time Setup” is the time both approaches use to get ready for testing. Here SkiRaff runs a DWPopulator, as explained in Section 6, this entails the running of the ETL program, and creating the DWRepresentation object used by the predicates. Whereas manual just runs the ETL program as is. ”Execution Time Test Cases” is the time each approach used on running all the test cases described above. Execution time is almost identical between the two cases, indicating that SkiRaff is at least as good as manual testing in this regard.

Table 1. Result of Evaluation

	SkiRaff	Manual
Number of statements	11 stmt	110 stmt
Execution Time Setup	79.52 sec	79.44 sec
Execution Time Test Cases	18.02 sec	18.23 sec
Execution Time Total	97.52 sec	97.67 sec

10. Conclusion and Future Work

In this paper we presented SkiRaff, an ETL test-framework for use in conjunction with pygrametl. We started off by looking into the field of ETL testing. We found that most other test frameworks either were not specialized in ETL testing, or they were GUI based. As pygrametl is API based, it was decided that SkiRaff should not be using a GUI and be API based as well, as it would otherwise be a clash in usage philosophy. By looking deeper into ETL testing, we found that its main purpose is to ensure correct data flow from sources to the DW. As this is a rather data-centric problem, we based SkiRaff upon source-to-target testing. Here testing of an ETL occurs by focusing on the DW that it populates.

SkiRaff supports automatic testing at the system level, and assists in both functional- and regression testing. It consists of two larger components, the DWPopulator and the Predicates. The DWPopulator allows for the execution of pygrametl programs, so that a DW may be populated for test. It also

allows testers to dynamically replace sources and DW in the pygrametl program's source code with test data. The Predicates contain the actual test functionality. There are different types of predicates, each allowing for the implementation of a common assertion on the DW's contents. Once a tester has set up all their predicates, they are executed. It is then reported on, whether their assertions held or not. If not, the cause of failure will be reported.

Having developed SkiRaff, we evaluated its use as compared to manual testing through SQL queries. We found that the execution speeds were almost equal when using SkiRaff. Yet, while manual testing needed 110 statement to be performed, testing with SkiRaff needed only 11 statements. This a benefit for manual testers in migrating to SkiRaff. Yet, we must remember that our predicates can only be used to cover a portion of an ETL system. It is likely that there are situations during testing, where the flexibility of a SQL query could come in handy. However, we determine that SkiRaff can be used to cover the most common cases during testing. As such SkiRaff, would be a viable tool in any pygrametl tester's toolkit.

If more work is to be done on SkiRaff, some functionality could be added. There is a restriction upon the types of programs, which the DWPopulator can dynamically replace sources and DW. Thus, an upgrade to this component would allow for more types of programs to be processed. More predicate types could be added to the framework in order to handle more specialized assertions. It would also be beneficial to implement features that allow testers to develop their own predicate types that run on SQL. As our current user-defined predicate types, RuleRowPredicate and RuleColumnPredicate, work through python functions and are as such slow to execute. Furthermore, a lot of SkiRaffs functionalities depend on it performing natural joins, which restricts the range DW schemas that it will function on. Allowing users to specify columns to perform normal joins on would increase this range. Thus it could be a valuable addition.

Acknowledgments

Thanks to our supervisor Christian Thomsen for his assistance during this project.

References

[1] Christian Thomsen and Torben Bach Pedersen. pygrametl: A powerful programming framework for extract-transform-load programmers. In *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*, pages 49–56. ACM, 2009.

[2] QuerySurge. Big data testing etl, testing & data warehouse testing. `http://www.querysurge.com/`, 2016.

[3] AnyDbTest. Anydbtest - home. `https://anydbtest.codeplex.com/`, 2011.

[4] Neveen ElGamal, Ali El Bastawissy, and Galal Galal-Edeen. Towards a data warehouse testing framework. In *ICT and Knowledge Engineering (ICT & Knowledge Engineering), 2011 9th International Conference on*, pages 65–71. IEEE, 2012.

[5] Subu Iyer. Enabling etl test automation in solution delivery systems. In *PNSQC 2014 Proceedings*, 2014.

[6] TN Manjunath, Ravindra S Hegadi, HK Yogish, RA Archana, and IM Umesh. A case study on regression test automation for data warehouse quality assurance. *International Journal of Information Technology*, 5(2):239–243, 2012.

[7] Sara B Dakrory, Tarek M Mahmoud, and Abdelmgeid A Ali. Automated etl testing on the data quality of a data warehouse. *International Journal of Computer Applications*, 2015.

[8] Christian Thomsen and Torben Bach Pedersen. Etldiff: a semi-automatic framework for regression test of etl software. *Lecture Notes in Computer Science*, 4081:1, 2006.

[9] Christian Thomsen, Ove Andersen, and Søren Kejser Jensen. pygrametl - etl programming in python. Internet, 2016. Accessed: 17-04-2016.

[10] Marc-André Lemburg. Pep 249 – python database api specification v2.0. Internet, 2016. lAccessed: 15-04-2016.

[11] A.M.J. Hass. *Guide to Advanced Software Testing*. IT Pro. Artech House, 2008.

[12] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[13] V. Rainardi. *Building a Data Warehouse: With Examples in SQL Server*. Apresspod Series. Apress, 2007.

[14] Brandborg Alexander, Kjær Arash Michael Sami, Jensen Mathias Claus, and Mikkelsen Mikael Vind. Skiraff git repository. `https://github.com/Betaboxguugi/SkiRaff`, 2016.

1. PygramETL Test Script

```
from setup_dw import mk_dw
from setup_inputs_large import mk_author_db, mk_book_db, mk_country_csv
import sqlite3
import pygrametl
from pygrametl.datasources import SQLSource, CSVSource
from pygrametl.tables import FactTable, Dimension, SlowlyChangingDimension, \
    SnowflakedDimension

dw_path = './dw.db'
author_path = './author.db'
book_path = './book.db'
country_path = './country.csv'

mk_dw(dw_path)
mk_author_db(author_path)
mk_book_db(book_path)
mk_country_csv(country_path)

# Connections
dw_conn = sqlite3.connect(dw_path)
author_conn = sqlite3.connect(author_path)
book_conn = sqlite3.connect(book_path)
country_handle = open(country_path, "r")

wrapper = pygrametl.ConnectionWrapper(dw_conn)

# Sources
author_src = SQLSource(connection=author_conn, query="SELECT * FROM author")
book_src = SQLSource(connection=book_conn, query="SELECT * FROM book")
country_src = CSVSource(f=country_handle, delimiter=',')


# Tables
author_dim = Dimension(
    name='authordim',
    key='aid',
    attributes=['name', 'city', 'cid'])

book_dim = SlowlyChangingDimension(
    name='bookdim',
    key='bid',
    attributes=['title', 'year', 'version'],
    lookupatts=['title'],
    versionatt='version')

country_dim = Dimension(
    name='countrydim',
    key='cid',
    attributes=['country'],
    lookupatts=['country'])

fact_table = FactTable(
    name='facttable',
        keyrefs=['aid', 'bid'])

snowflake = SnowflakedDimension([(author_dim, country_dim)])

# We map cities to countries and populate the countrydim
cid_map = {}
for row in country_src:
    cid = (country_dim.ensure(row))
    cid_map[row['city']] = cid

# We populate the authordim and the fact table
for row in author_src:
    if row['city'] in ['Hadsten','Skanderborg','Kobenhavn']:
        row['cid'] = cid_map[row['city']]
    else:
        row['cid'] = None
    row['name'] = row['firstname'] + ' ' + row['lastname']
```

Code Example 21. PygramETL Test Script

Continued next page

Continued from last page

```
    row['name'] = row['firstname'] + ' ' + row['lastname']
    row.pop('aid', 0) # Gets rid of aid so that pygrametl can generate them

    # Placing new row in author_dim
    row['aid'] = author_dim.ensure(row)

    # Placing new row in fact_table
    fact_table.ensure(row)

# Places books directly into book_dim
for row in book_src:
    book_dim.scdensure(row)

wrapper.commit()
wrapper.close()

author_conn.close()
book_conn.close()
country_handle.close()
```

Code Example 22. PygramETL Test Script

2. SkiRaff Test Script

```
from framework.case import Case
from framework.dw_populator import DWPopulator
from framework.predicates import *
import sqlite3
import time

start = time.monotonic()

def time_passed(start_time):
    end = time.monotonic()
    elapsed = end - start_time
    return '{}{}'.format(round(elapsed, 3), 's')

table1 = 'authordim'
table2 = 'bookdim'
table3 = 'countrydim'
fact_table = 'facttable'
goodbooks = 'goodbooksdim'

cnnp_test = ColumnNotNullPredicate(table1)
ctp_test = CompareTablePredicate(table2, goodbooks, ['ID'], True, False, (), True, True)
fdp_test = FunctionalDependencyPredicate([table1, table3], 'cid', 'city')
ndrp_test = NoDuplicateRowPredicate(table1, ['city', 'aid'], True)
rip_test = ReferentialIntegrityPredicate()
rocp_test = RowCountPredicate(table2, 6)
scdvp_test = SCDVersionPredicate(table2, {"title": 'EZ PZ ETL'}, 4)

pred_list = [cnnp_test, ctp_test, fdp_test, ndrp_test, rip_test, rocp_test,
             scdvp_test]

dw_path = './dw.db'
pygrametl_program_path = './etl.py'
dwp = DWPopulator(pygrametl_program_path, sqlite3, True, database=dw_path)

dw_rep = dwp.run()

# Checking how long it took.
time_before_test = time_passed(start)

case = Case(dw_rep, pred_list)

case.run()


time_after_test = time_passed(start)

# Checking how long it took.
print(" TIME BEFORE TEST " + time_before_test)
print(" TIME AFTER TEST " + time_after_test)
```

Code Example 23. SkiRaff Test Script

3. Manual Test Script

```
import sqlite3
import time

start = time.monotonic()

def time_passed(start_time):
    end = time.monotonic()
    elapsed = end - start_time
    return '{}{}'.format(round(elapsed, 3), 's')

# We run the ETL program
etl_path = './etl.py'
with open(etl_path, 'r') as f:
    code = compile(f.read(), etl_path, 'exec')
    exec(code)

dw_path = './dw.db'
conn = sqlite3.connect(dw_path)
c = conn.cursor()

time_before_test = time_passed(start)
print(time_before_test)

# Not Null
c.execute("""SELECT *
             FROM authordim
             WHERE aid IS NULL
               OR city IS NULL
               OR name IS NULL
               OR cid  IS NULL""")
not_null_list = c.fetchall()
if not_null_list:
    print('Null found on the following elements:')
    for row in not_null_list:
        print(row)
else:
    print('Not Null Test - Success')

# Compare Table
c.execute("""SELECT *
             FROM goodbooksdim
             EXCEPT
                SELECT *
                FROM bookdim """)
some_list = c.fetchall()
if some_list:
    print('Failed to compare tables')
    for row in some_list:
        print(row)
else:
    print('Successfully compared tables')

# Functional Dependency
c.execute("""SELECT DISTINCT t1.cid ,t1.city
             FROM (authordim NATURAL JOIN countrydim) as t1 ,
                  (authordim NATURAL JOIN countrydim) as t2
             WHERE  t1.cid = t2.cid  AND (  (t1.city <> t2.city)  )""")
func_dep_list = c.fetchall()
if func_dep_list:
    print('Functional Dependency did not hold on the following elements:')
    for row in func_dep_list:
        print(row)
else:
    print('Functional Dependency Test - Success')

# No Duplicate Row
```

Code Example 24. Manual Test Script

Continued next page

Continued from last page

```
# No Duplicate Row
c.execute(""" SELECT name,city,aid,cid ,COUNT(*)
              FROM authordim
              GROUP BY name,cid HAVING COUNT(*) > 1 """)
some_list = c.fetchall()
if some_list:
    print('Duplicates found on the following elements:')
    for row in some_list:
        print(row)
else:
    print('No Duplicate Row Test - Success')

# Referential Integrity
c.execute("""SELECT *
             FROM authordim
             WHERE NOT EXISTS(
                SELECT NULL
                FROM countrydim
                WHERE authordim.cid = countrydim.cid)""")
ref_integ_list = c.fetchall()
c.execute("""SELECT *
             FROM countrydim
             WHERE NOT EXISTS(
                SELECT NULL
                FROM authordim
                WHERE countrydim.cid = authordim.cid)""")
ref_integ_list += c.fetchall()
c.execute("""SELECT *
             FROM facttable
             WHERE NOT EXISTS(
                SELECT NULL
                FROM authordim
                WHERE facttable.aid = authordim.aid)""")
ref_integ_list += c.fetchall()
c.execute("""SELECT *
             FROM authordim
             WHERE NOT EXISTS(
                SELECT NULL
                FROM facttable
                WHERE authordim.aid = facttable.aid)""")
ref_integ_list += c.fetchall()
c.execute("""SELECT *
             FROM facttable
             WHERE NOT EXISTS(
                SELECT NULL
                FROM bookdim
                WHERE facttable.bid = bookdim.bid)""")
ref_integ_list += c.fetchall()
c.execute("""SELECT *
             FROM bookdim
             WHERE NOT EXISTS(
                SELECT NULL
                FROM facttable
                WHERE bookdim.bid = facttable.bid)""")
ref_integ_list += c.fetchall()
if some_list:
    print('Referential Integrity did not hold on the following elements')
    for row in some_list:
        print(row)
else:
    print('Referential Integrity Test - Success')

# Row Count
c.execute("""SELECT COUNT(*)
             FROM bookdim """)
row_count = c.fetchall()[0]
if row_count == 6:
    print('Row Count Test - Success')
else:
    print('Row count did not hold\nThere are {} rows when there should be 6')

# SCD Version
```

Code Example 25. Manual Test Script

Continued next page

Continued from last page

```
# SCD Version
c.execute("""SELECT max(version)
             FROM bookdim
             WHERE title = 'EZ PZ ETL'""")
scdv_list = c.fetchall()
if scdv_list[0] == 4:
    print('SCD Version Test - Success')
else:
    print('SCD Version did not hold. Should have been 4 but was ' +
          str(scdv_list[0][0]))


time_after_test = time_passed(start)
# Checking how long it took.
print(" TIME BEFORE TEST " + time_before_test)
print(" TIME AFTER TEST " + time_after_test)
```

Code Example 26. Manual Test Script