SUMMARY

This paper presents DeclarativeETL, an addition to PygramETL that allows developers to generate PygramETL in Python and data warehouse in Data Definition Language, from a shared specification. This generated code is to be used as a starting point for data scientists wanting to set up an Extract-Transform-Load (ETL) program. The generated PygramETL objects matches the generated data warehouse tables, such that a connection may be established and PygramETL can interact with the data warehouse, inserting data into tables.

DeclarativeETL uses declarative specifications that are written in Tom's Obvious Minimal Language (TOML). TOML is a configuration language used to define settings as key/value pairs under common names. TOML is used to declare dimensions and fact tables, each being an array of table members, i.e. attributes and measures. In conjunction with tables, a set of defaults must be declared to save the developer from specifying common values. A description of the default values follows: Which database management system should be used for the data warehouse. The name of the data warehouse, used in creation of a Data Definition Language (DDL) script and connection to the data warehouse from PygramETL. The naming convention used to define primary keys, allowing them to be declared implicitly based on table name and the convention. The schema type, allowing for the implicit creation of fact table foreign keys. The column types for dimension attributes and fact table measures, this default value may be overridden at an individual level, when declaring attributes or measures.

Dimension are declared by a new field, e.g. [dimension.name]. Anything declared after this point will belong to that dimension, until a new is declared. Attributes and roles can be assigned by supplying an array of names, e.g. attributes = [name, address], where the default type can be overridden by specifying a new type using "name: type". Attributes are used to create table columns, and roles are used to create multiple fact table references to a dimension. The names assigned are used in conjunction with the default type to generate DDL and PygramETL code for the dimensions name, attributes, and primary key.

Fact tables are declared similarly as [fact.name], but assigned measures instead. The measures are assigned identically to the attributes. A fact table requires foreign keys to the related dimensions, for a Star schema this is all dimensions in the same group as the fact table, or in the case of no groups all the dimensions declared. The foreign keys are based on the implicit primary keys, unless roles are specified, in which case the role names are used.

Groups are declared by prepending group.name to a dimension or fact table, such that it becomes [group.$groupname$.dimension.$name$], where the names are up to the user.

In general the syntax of a declarative specification followed: First a set of defaults. Then a combination of groups, dimensions and fact tables, but at least one dimension and fact table, each with one or more attributes/roles/measures.

DeclarativeETL was implemented in Python by using the `tomllib` package to initially parse the declarative specification into a Python dictionary. The dictionary was then further parsed into specification, dimension and fact table objects to simplify the process of generating code. A specification object, much like a declarative specification, contains the default values and each dimension and fact table. The then parsed objects are used to generate string segments for tables in DDL and PygramETL, using a template for tables. The template contained the static code for a table, and was populated with the name, keys and columns of the table.

In addition to these string segments DDL code required initial code to create the database to contain the tables. Similarly, for the generated Python code to be correct it was required to import the necessary packages and create a connection to the data warehouse, so that PygramETL objects may interact with it. Once this initial code was written to file, each table string segment could be written. Code generation was complete when all tables had benn written to file.

DeclarativeETL was evaluated in regards to developer productivity in terms of lines and characters used to declare tables. DeclarativeETL was also evaluated in terms of runtime and correctness. Runtime analysis proved that that the execution time and memory usage was negligible and had little to no impact on the user or machine, as it resulted in an average of 26.2ms of execution time and 22.2 MiB RAM. Correctness was evaluated by executing the generated DDL file in PostgreSQLs psql, and the generated Python file using Python 3.11 with the latest version PygramETL. Both files executed without trouble and gave the expected result.

Productivity was measured using a dimension with 2 and 10 attributes, respectively. The evaluation for the dimension with 2 attributes resulted in a decrease in lines written by 75%, and characters written by 77%. For the dimension with 10 attributes these values were 87.5% and 68% for lines and characters, respectively. Showing that even as dimensions increase in size, better productivity is maintained.

Declarative Data Warehouse setup in PygramETL

Simon Mathiasen *Aalborg University*
smath17@student.aau.dk

***Abstract*—In order to begin Extract-Transform-Load programming a data warehouse must be created in a database management system, and the schema of the data warehouse must be programmed in an Extract-Transform-Load framework to properly load data from sources. However, the set up of a data warehouse and the definition of a schema in an appropriate framework can be labor intensive. Furthermore, the complexity of this task increases as schemas become bigger, as the developer must ensure that the data warehouse schema matches the schema defined in the framework for Extract-Transform-Load. In this paper I present the framework DeclarativeETL which is an addition to PygramETL [3] used to generate implementation for data warehouse schema, and PygramETL. DeclarativeETL results in a DDL and Python file generated from a shared declarative specification. By exploiting TOML [6], a simple configuration language, and a simple syntax for the declarative specification, developer productivity is increased as they are only required to name dimension and fact tables, and their respective attributes and measures, in conjunction with a set of default values, e.g. schema type, and attribute and measure types. The defaults saves the developer many keystrokes as most attributes share the same type. DeclarativeETL is evaluated to be fast and lightweight while providing more than 100% increased productivity in terms of lines of code when compared to programming DDL/PygramETL manually.**

***Index Terms*—Data warehouse, Extract-Transform-Load, Declarative Programming, Code generation, PygramETL**

I. Introduction

Setting up a DW is a non-trivial task, requiring a developer to design a database schema and ETL flow, then implement it for both ETL framework and Database Management System (DBMS) of choice, i.e. PygramETL and PostgreSQL, respectively. However, these implementations must be done in different languages, e.g. Python for PygramETL and Data Definition Language (DDL) for Postgresql. This is not ideal given that they are both based on the same domain and as such describe the same in different syntax. It would be simpler if both implementations could be generated from a single, joint design specification. In this paper I will present an addition to PygramETL which will let developers generate Python and DDL implementations for the setup of dimensions and fact tables for PygramETL and DBMS from a declarative specification. This will not only increase developer productivity, but also reduce developer errors, making sure both implementations are identical to the specification.

To further increase developer productivity, the specification is declared in an easy to read and write configuration language. This will allow the developer to simply declare the core of their specification, i.e. measures, dimensions and respective attributes, in a simple format. The specification will also contain a set of default values, such as types and schema type used to populate the tables.

A simplified example use case is seen in **Figure 1**. The example is based on Star Schema Benchmark (SSB) [5] and is used to store and track transactions. Each lineorder stored in the fact table describes a transaction in which a customer ordered a certain part. Each fact is connected to Part, Customer and Date dimension. The use case is modeled as a specification in Listing 1.

The addition to PygramETL, named DeclarativeETL, presented in this paper enables developers to create such a declarative specification which in turn increases developer effectiveness due to less code required and less errors, given the convenience of identical implementations of database schema and ETL schema.

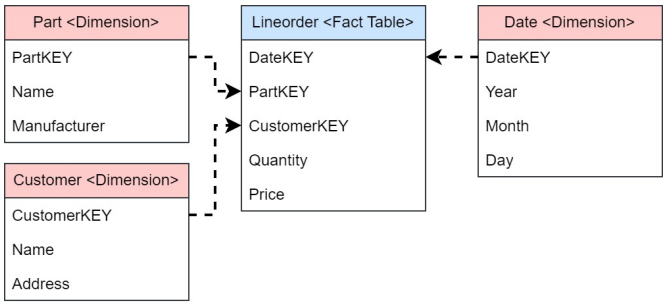

Fig. 1. SSB use case example (simplified)

Previous efforts have been made in increasing productivity for data warehousing, such as SimpleETL [4] and PygramETL [3]. However, these efforts use programmatic approaches. Where as DeclarativeETL uses a declarative approach, in which database tables can be declared without the use of programming, by instead listing their names and respective members in a configuration file. This is the main contribution of DeclarativeETL which grants increased productivity.

The rest of this paper is structured as follows: First a brief description of the problem domain in **Section II**. Then related work is discussed in **Section III**. Followed by a description of a DeclarativeETL specification in **Section IV** showcasing both its content and the grammar used to write it. Then in **Section V** the implementation of DeclarativeETL is explained in regards to parsing the specification and generating code. In **Section VI** DeclarativeETL is tested and evaluated in terms of productivity. Finally **Section VII** concludes the paper and suggests areas for future work.

II. Background

A Data Warehouse (DW) is a data storage system that aggregates huge volumes of data from different sources into a single consistent repository of data for analytical purposes

[1]. A DW utilizes the Extract, Transform, Load (ETL) process [2], in which data is first extracted and collected from multiple different sources into a staging area. ETL frameworks are usually used to setup and automate this, as they simplify the process by providing common functionality, one such framework is PygramETL [3] which specializes in easy and efficient ETL in Python. Data in the staging area is then processed and transformed depending on the use case, such as the data being filtered, de-duplicated, validated, and formatted to fit the schema of the DW. For this paper only relational schemas are considered. A relational schema, such as Star or Snowflake, is the logical description of a DW and defines how the data is organized into tables in a relational database. Lastly, the now transformed data is loaded from the staging area and into the data warehouse. As mentioned a DW has a schema defining tables and their names, attributes, types, measures, etc. and the relation between them. In a schema data is categorized into structures, dimensions and fact tables. Fact tables contain facts that consist of measures of a business process. Consider that process to be selling replacement parts, then each fact is an order, with measurements such as quantity and price. Measurements are commonly of numeric additive types. Dimensions hold attributes that describe a fact and is commonly of text or date types. Following the previous example, an order can be described by the customer who made the order, or the date of when the order was placed. Two common types of relational DW schemas are the Star schema and the Snowflake schema [1]. A Star schema consist of a single and central fact table connected to surrounding dimension tables, one per dimension, and hence forming a star structure. A Snowflake schema is similar, with the exception that each dimension may have multiple dimension tables, one for each level in the dimension. These dimension tables are connected by a many-to-one relation, hence forming a snowflake-like formation. This type of schema supports and renders dimension hierarchies, in which a dimension consist of multiple dimension tables, one per level in the hierarchy, with a relation to the level above it. A level represents a level of detail, the lower the more specific and detailed, e.g. ”Year, Month, Day” for a date dimension, where ”Day” is the lowest level. From the lowest level more details can be gathered by ascending in level. Each of these hierarchies has a single relation to the fact table, at the lowest level [1].

III. Related work

Previous efforts of increasing productivity for data warehousing has been focused on ETL flow, such as PygramETL [3], and also the set up of the data warehouse, such as SimpleETL [4]. Both of these approaches propose a framework in which the developer must write Python code, in contrast DeclarativeETL proposes an addition to PygramETL, in which the developer can create a specification of the structure of their data warehouse and ETL using an easy to read and write configuration language. The specification is then used to generate a database script and a PygramETL file containing objects for all tables.

PygramETL proposes a Python package for the creation of ETL flows in Python. Using Pythons design philosophy for high-level abstractions and its productive style. Productive in the sense that it requires far fewer keystrokes to write Python code compared to other OOP-languages such as Java. PygramETL allows for the creation of tables and connection to their database counterpart. However, not the creation of the database counterpart itself. PygramETL also allows creating transformation rules and extraction/insertion of data from a data source into a data warehouse.

SimpleETL proposes a framework in which the developer may program a specification of their data warehouse in Python. SimpleETL uses the specification to generate SQL for the creation or altering of a data warehouse schema, and required PygramETL objects to interact with the schema. This is very similar to the goals of DeclarativeETL, again, with the exception that it is done entirely in a simpler format, to create a starting point for ETL programming. Given the simpler format DeclarativeETL requires even less experience than SimpleETL

Both PygramETL and SimpleETL are programmatic, meaning the developer must write their respective programs using objects and describing most of the steps in code. Where as DeclarativeETL, is designed to be declarative, meaning that tables can simply be declared, and then implicitly used to generate SQL for the data warehouse and Python for ETL flows.

IV. Declarative Specification

This sections explains each part of a declarative specification used to set up a Data Warehouse and is based on the use case example from **Figure 1**

A declarative specification is a configuration of the developers data warehouse schema, describing the logical representation of dimension tables and fact tables, the relations between them, as well as their attributes, measures and types. An example of a declarative specification based on the aforementioned use case can be seen in **Listing 1**. A declarative specification for the full SSB can be found in Appendix A. In order to make the specification as simple as possible, it is written in a easy to read and write language, for this Tom’s Obvious Minimal Language (TOML) [6] is used. The specification consists firstly of defaults defining a standard for the schema, secondly a combination of dimensions and fact tables, each with their respective attributes or measures,. A breakdown of the different parts are as follows:

A. Defaults

The defaults are used to specify the standards of the declarative specification, this allows for a simple specification with a high level of abstraction which leads to less code, less errors and fewer repetitions. All this increases developer productivity in terms of effectiveness and convenience. In the example declarative specification from **Listing 1** the defaults (lines 2-8) specify which Database Management System (DBMS) is going to be used, this is required since the DBMS sets the ground rules for how the database should be created. One of the elements the DBMS defines is the Data Definition Language (DDL) which is used to create the database schema. The second value is the name used for the data warehouse, this

value is used to create the database in the DBMS, but also for PygramETL to connect to it. Then comes the naming convention for primary keys (PK), this value can save developers from specifying PK names themselves. Given by `pk_naming`, this is what is to be concatenated onto the name of the dimension to create the PK of that dimension, given the value ”KEY” the resulting PK for the Customer dimension is ”CustomerKEY”. Next the schema type i.e. star or snowflake, is specified. Both schema types allows for implicit creation of foreign keys (FK) between dimension and the fact table. However, the schema type determines how the implicit creation should be carried out. For a star schema each dimension’s primary key can be added to the fact table as a foreign key. For a snowflake schema the same can be done, but for the relational implementation of the dimensions only the lowest level table per dimension hierarchy is connected to the fact table. This will be be further explained in **Section IV-B**. Lastly, field type defaults are specified for attributes and measures, respectively. The type specified must be valid for the chosen DBMS. Default field types leads to a much simpler specification as for any attribute or measure following the default only requires to be named. For example, if `dimension_attribute_type` is set to `VARCHAR(30)` then any non-key attribute in all the dimensions will have the type `VARCHAR(30)`, such as ”Name” and ”Address” for the Customer dimension. `fact_measure_type = DECIMAL(10, 4)` functions the same, but applies to all measures in all fact tables. The types can be overridden for each attribute or measure at an individual level when specified. A colon (:) is used to separate the name and type of a attribute or measure as seen in line 21 of **Listing 1**. In the example ”quantity” will be of type of INTEGER and ”price” will be of type DECIMAL(10, 4).

```
1  # Declare defaults
2  [Default]
3  DBMS = "postgresSQL 15.2"
4  data_warehouse_name = "test"
5  pk_naming = "KEY"
6  schema_type = "star"
7  dimension_attribute_type = "VARCHAR(30)"
8  fact_measure_type = "DECIMAL(10, 4)"
9
10 # Declare each dimension
11 [dimension.Customer]
12 attributes = ["name", "address"]
13
14 [dimension.Part]
15 attributes = ["name", "manufacturer"]
16
17 [dimension.Date]
18 attributes = ["day: INT", "month: INT", "year: INT
       "]
19
20 [fact.Lineorder]
21 measures = ["quantity: INT", "price"]
```

Listing 1. Example of schema specification

B. Dimensions

Each dimension must be declared in their own section, denoted by brackets as [dimension.name]. However, given that the PK can be generated and the types are default, the developer only has to specify the name of the dimension (which is also the name of the configuration section) and the names of the non-key attributes. As seen in **Listing 1** lines 10-18, a dimension is declared by its name in brackets, followed by a list of attributes and possible type overrides, as seen in line 18.

C. Fact Tables

Declaring a fact table is very similar to that of a dimension. As fact tables are declared in sections using [fact.name]. Declaration of measure is similar to that of attributes, as seen on line 21 in **Listing 1**.

D. Groups

A group is used to create relations between fact tables and dimensions. For a Star schema the fact table will have a relation to any dimension in its group. Groups are declared by prepending group.$group-name$ to any dimension or fact table, resulting in e.g. [group.Orders.fact.Lineorders], for the group ”Orders” and fact table ”Lineorders”. If no group is specified it is implicitly assumed that all tables are in the same group.

E. Dimension Roles

Dimension can play roles. By playing roles the dimension can appear multiple times, by different keys, in the same fact table. When a dimension plays roles, the fact table must have multiple foreign keys referencing the dimension.

In DeclarativeETL role-playing dimensions are declared by specifying its roles as the names it will be referenced by. We use roles to expand on the Date dimension in the running example. The new Date dimension can be seen in Listing 2. In the example the fact table will then implicitly use the role names as foreign keys, instead of the dimensions’ primary key.

```
1 [dimension.Date]
2 18 attributes = ["day: INT", "month: INT", "year:
      INT"]
3 roles = ["CommitDate", "ShippingDate"]
```

Listing 2. Date dimension expanded with roles

F. Grammar

The grammer for DeclarativeETL builds on TOML and its grammar, which can be found at https://toml.io/en/v1.0.0. The main structures of TOML are key/value pairs, declared by the syntax ”key = value” and that every pair must appear on a separate line. However, multiple values can be assigned to a key by the use of arrays such as ”key = [value1, value2]”, this is especially useful for specifying attributes and measures. DeclarativeETL builds on this grammar, but expects all key/value pairs to be encapsulated in TOML-tables [7], such as the default values which must be placed within the $[default]$ TOML-table. DeclarativeETL will not parse anything outside TOML-tables. A TOML-table acts like a dictionary in that key/value pairs are stored under a

common name defined by a header enclosed in brackets, e.g. `[table-1]`, the scope of such table lasts until the next table or end-of-file (EOF). TOML-tables are used in DeclarativeETL to encapsulate objects, and nested TOML-tables are used to declare groups e.g. [group.group-name.(dimension/fact).name]. The group will encapsulate the sub dimension or fact TOML-table, as such it acts like a TOML-table of TOML-tables.

TABLE I
SYNTACTIC CATEGORIES

Symbol		Abbreviation	Description
S	$\in$	**Spec**	Specification
D_{DEF}	$\in$	**DecDef**	Default Declarations
D_{DIM}	$\in$	**DecDim**	Dimension Declarations
D_F	$\in$	**DecFact**	Fact table Declarations
AM	$\in$	**Att**	Attributes and Measures
t	$\in$	**Type**	Data Types in string form
s	$\in$	**Str**	String Literals

TABLE II
PRODUCTION RULES

S	::=	D_{DEF} $(G \mid D_F \mid D_{DIM})$+
D_{DEF}	::=	[default] DBMS = "s" data_warehouse_name = "s" pk_naming = "s" schema_type = "t" dimension_attribute_type = "t" fact_measure_type = "t"
D_{DIM}	::=	[(group.s.)? dimension.s] attributes = [$AM(,\ AM)*$] $\big($roles = ["s"$(,\ $"$s$"$)*\big)$?
D_F	::=	[(group.s.)? fact.s] measures = [$AM(,\ AM)*$]
AM	::=	"$s : t$" $\mid$ "s"

The grammar used to create a declarative specification with DeclarativeETL is two-part and consist firstly of syntactic categories (**Table I**), describing the terminal and nonterminal symbols of DeclarativeETL and secondly a set of production rules (**Table II**), which are used to recursively replace nonterminal symbols, until only terminals remain. Terminal symbols are shown using lowercase letters, these are literals and cannot be changed using production rules. The terminal symbols of DeclarativeETL are data types (t), and string literals (s). Note that data types must be written as strings to conform with TOML, but is given a separate category as the data type must also exist in the specified DBMS. It will therefore cause an error unless the data type is either standard or user-defined in the DBMS.

Unlike the terminal symbols, the nonterminal symbols are shown using uppercase letters which are to be recursively replaced using the production rules. The production rules use extra notation, namely $+$, $*$, $?$, $|$. A plus symbol ($+$) means the nonterminal must be present one or more times, a multiply symbol $*$ means the nonterminal can be present zero or more times, a question mark $?$ means the rule is optional as it can appear zero or one time, a vertical line ($|$) is used to indicate an "or" meaning the AM rule can become either $s : t$ or s. Parentheses can be used to group symbols under these special notations, this also means that any parentheses seen in the grammar is not to be written, unlike the brackets, colons and periods which are literals part of the grammar.

The start rule (S) defines a declarative specification and can be replaced by a default table followed by at least one group, fact table, or dimension.

V. IMPLEMENTATION

It is important to note that as of this paper only the Star schema type is supported. The following sections are therefore only concerned with implementation related to Star schemas. DeclarativeETL is implemented in Python and packaged with PygramETL [3], meaning it is available if the `pygrametl` package is installed. DeclarativeETL is made of three components namely Parsing, DDL- and PygramETL code generation. An overview of these can be seen in **Figure 2**. The developer must input a specification, which is parsed and transformed into Python objects. These objects are used to generate DDL and PygramETL code. The resulting DDL and PygramETL files, are then to be used to setup database and ETL flow respectively. It uses the tomllib [8] TOML parser from the Python standard library, for Python version 3.11+. `tomllib` is used to parse TOML formatted Python File Objects into a Python dictionary, given a file path. The dictionary will contain sub-dictionaries for each TOML-table declared, meaning the default values and each group, dimension or fact table. Each sub-dictionary (default, group, dimension, fact table) contains the key/value pairs that describe them. A group dictionary contains more dictionaries for any dimension or fact table it encapsulates.

The DDL and Python files generated from the running example specification in **Listing 1** and **Listing 2** can be found in Appendix B.

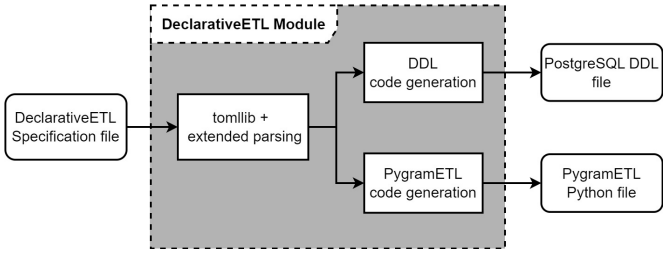

Fig. 2. Component diagram of DeclarativeETL and input/output

A. Parsing

To lessen the burden generating code directly from the dictionary, it is further parsed and transformed into a specification object. A specification object contains the default values, a set of dimension and fact tables and any group that may be present in the original specification. The `tomllib` dictionary is transformed into an `IntermediateSpecification` object. A class diagram of `IntermediateSpecification` can be seen in **Figure 3**, it shows that a `IntermediateSpecification` contains fields for all default values, it also consists of one or more `ParsedFactTable` and one or more

`ParsedDimension`. These are object versions of their dictionary representation, obtained by parsing it.

`ParsedFactTable` has a single field, a list that holds the names and key references of dimensions connected to the fact table. The key reference is the primary key of the dimension unless it plays roles. Similarly, `ParsedDimension` has to fields to hold its primary key and any roles it may play.

`ParsedFactTable` and `ParsedDimension` are both specializations of `ParsedTable`, which contains the table name and zero or more `ParsedAttribute`, that is used to represent both measures and attributes. A `ParsedAttribute` contains the attribute name and type, if a type is not specified for the individual attribute, the default is used.

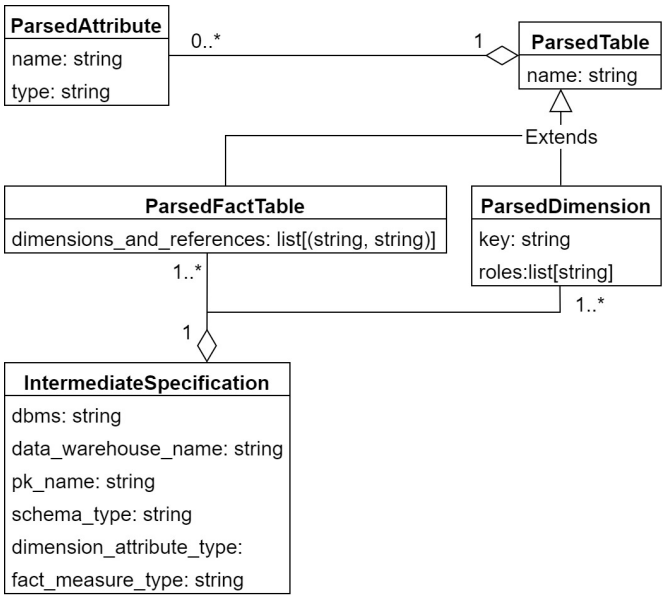

Fig. 3. Class diagram of IntermediateSpecification

The `tomllib` dictionary is parsed by first reading the values of the default sub-dictionary by using the keys of required values, e.g. `dbms = default.get("DBMS")`, where `default = tomllib_dict["Default"]`.

The content of each group is then parsed and added to the `IntermediateSpecification` object, encapsulating the now parsed dimensions and fact tables. However, before groups, dimensions or fact tables can be parsed, the sub-dictionary of those must be cast into a view object, given that key to extract each group, dimension or fact table (their name) is not known. The view object contains all key/value pairs in the dictionary, including the key for the dictionary itself and is obtained by `tomllib_dict.get("group").items()`, for groups. Given the view object all information becomes available. However, to make it iterable, the view object is cast to a list. An concrete example of such a list is:

$$[('tablename', \{\{'attributes' : name, age, sex'\}, \{'roles' : age'\}\})]$$

This results in a list of string/dictionary pairs formatted as $[('group/table - name', \{content\})]$, where $[content]$ is a dictionary containing dimensions and fact tables for groups, and attributes and measure for dimensions and fact tables, respectively, each of these are themselves a dictionary within $[content]$. To obtain the dimensions and fact tables of a group, its $[content]$ dictionary is handled as previously mentioned, where it is first cast into list of a view object. When parsing the groups, they and their content are removed from the dictionary (using the `pop` method instead of `get`), to avoid duplicate dimensions and fact tables, and so that the remaining (those declared before any group) dimensions and fact tables can be used to essentially form their own group in regard to which dimensions connect to the fact table.

When a dimension is transformed from dictionary to `ParsedDimension`, it is cast into a list of a view object as previously mentioned, from this list the name, roles, and attributes (and possible types) of the dimension is collected and used, together with instance variables "PK_NAME" and "DIMENSION_ATTRIBUTE_TYPE", to instantiate a new `ParsedDimension` object which is then added to the `IntermediateSpecification`. The instance variables are from the `IntermediateSpecification` object and cover some of the default values. The attributes are at this time parsed to become a pair of name/type represented by a `ParsedAttribute` object, if a type is specified by "name: type" it will override the default value, otherwise the default it used, hence why it is passed along. Returning to the dimension, a PK name is made by using $dimension.name+$ "PK_NAME", where "PK_NAME" is the default value for primary key naming convention. An algorithm for parsing a dimension is seen in **Algorithm 1**.

Algorithm 1: Method for parsing a dimension

Data: list: **DIM**
Instance variables: PK_NAME, DIMENSION_ATTRIBUTE_TYPE
Result: `ParsedDimension`

1 $name \leftarrow$ **DIM**[0]
2 $content \leftarrow$ **DIM**[1]
3 $attributes \leftarrow content["attributes"]$
4 $roles \leftarrow content["roles"]$
5 $pk \leftarrow name$ + PK_NAME
6 **return** *ParsedDimension(name, attributes, roles, pk, DIMENSION_ATTRIBUTE_TYPE)*

A similar set of actions are required to transform a fact table from dictionary to a `ParsedFactTable` object. The main differences being the keys used to extract information from the `[content]` dictionary, and key references (foreign keys) to the appropriate dimensions (dimension from the same group for Star schemas), to assign these references, the instantiating of a `ParsedFactTable` requires a list of related `ParsedDimensions`. The references are stored as $(name, reference)$ pair, where $name$ is the name of the dimension and $reference$ being either the dimensions' role or PK, depending on if the dimension plays any role. A pair is stored for each key or role.

Code generation is two part, as DDL code is generated to

be used for setting up of a database as the data warehouse, and PygramETL Python code is generated to be used as a starting template for an ETL flow.

B. DDL code generation

For DDL generation the default 'DBMS' value is used to determine the SQL dialect to generate DDL in. As of this paper only PostgreSQL is supported, with more possible in the future, as such this chapter will be specific to PostgreSQL. As the goal of the generated DDL file is to be executable it begins with "CREATE DATABASE dw_name; \connect dw_name" in order to create a database to hold the tables, this database is also in PygramETL, as it must connect to the database containing the data warehouse. The primary objective of DDL generation is to generate and write `CREATE TABLE` statements to a file. Generating these statements is done using the algorithms seen in **Algorithm 2** and **Algorithm 3**.

The algorithm (including string template) used when generating dimensions can be seen in **Algorithm 2**, the string template is seen in line 10. The algorithm, given an `IntermediateSpecification` object, will return a list of strings, containing a `Create Table` statement [9] string for each dimension in the specification. A single statement for a dimension is generated by first preparing a string for the primary key, this is done by concatenating the dimensions' key name with the string "SERIAL PRIMARY KEY" (line 4) in order to define the key name as the primary key with the data type serial. Secondly, the columns of the table must be prepared by cast all members to string, as the members are `ParsedAttributes` their string method returns the format "name: type". The column strings are joined into a single string using ",\n" as the separator (lines 6-9), as this will ensure each column appears on its own line to make the DDL file more readable. Finally, the string template can be populated with the values for the dimensions name, primary key and columns (line 10). The name is concatenated with "_Dimension" again to make the DDL file more readable, then the primary key is written as the first part of the statements body, followed by the columns. A populated example for the Customer dimension can be seen in **Listing 3**.

```
1  -- DeclarativeETL
2  [Default]
3  DBMS = "postgresSQL 15.2"
4  pk_naming = "KEY"
5  dimension_attribute_type = "VARCHAR(30)"
6
7  [dimension.Customer]
8  attributes = ["name", "address"]
9
10 -- DDL
11 CREATE TABLE Customer_Dimension
12 (
13 CustomerKEY SERIAL PRIMARY KEY,
14 name VARCHAR(30),
15 address VARCHAR(30)
16 );
```

Listing 3. Generated DDL dimension table

For fact tables, the main difference is that it has multiple foreign keys and the PK is a composite key of all the foreign keys. Similar to dimensions **Algorithm 3** is used to generate all fact table DDL statements of a specification.

Algorithm 2: DDL dimension generation

Data: `IntermediateSpecification`: **IS**
Result: List of DDL create table strings

1 $ddl\_tables \leftarrow []$
2 **foreach** $dim \in$ ***IS***$.dimensions$ **do**
3 $\quad pk \leftarrow dim.key + "SERIAL\ PRIMARY\ KEY"$
4 $\quad attributes \leftarrow []$
$\quad$ `// Cast each attribute into a string`
5 $\quad$ **foreach** $member \in dim.members$ **do**
6 $\quad\quad attributes \leftarrow attributes \cup string(member)$
7 $\quad$ **end**
$\quad$ `// Join strings using ",\n"`
8 $\quad column\_string \leftarrow attributes.join(",\n")$
$\quad$ `// Create string with linebreaks`
9 $\quad table\_statement \leftarrow$ ""
CREATE TABLE {$name$}_Dimension (
{pk},
{$column\_string$}
);""
10 $\quad ddl\_tables \leftarrow ddl\_tables \cup table\_statement$
11 **end**
12 **return** ddl_tables

The algorithm generates a single statement by first preparing the primary key of the format "PRIMARY KEY ($references$)" references to the dimension are collected (line 7) and used to populate the PK format, e.g. "PRIMARY KEY (CustomerKEY, PartKEY, CommitDate, ShippingDate)".

At the same time foreign keys are prepared (line 8, 9 and 12), they must adhere to the format "$reference$ SERIAL REFERENCES $dimension_name$_Dimension", becoming e.g. "CustomerKEY SERIAL REFERENCES Customer_Dimension". Serial is again used as the data type for keys, while "REFERENCES" is used to link the key to a table, "_Dimension" is required as it was used to make table names more readable. Once the primary and foreign keys are formatted, the measures of the fact table are processed and formatted (lines 13-16) identically to the attributes of the dimension. Lastly, the string template is populated to create a single string statement, an example of this can be seen in **Listing 4**.

Each fact table is processed and appended to at list. When all dimensions and fact tables have been processed, the generated statements can be written to a file, a minimal example of a generated DDL file can be seen in **Listing 5**.

Algorithm 3: DDL fact table generation

Data: `IntermediateSpecification`: **IS**
Result: List of DDL create table strings

```
ddl_tables ← []
foreach fact_table ∈ IS.fact_tables do
    // Begin pk string
    pk ← "PRIMARY KEY ("
    fkeys = []
    measures ← []
    foreach name, reference ∈ fact_table.dimensions_and_references do
        pk ← pk + "SERIAL PRIMARY KEY"
        fk ← reference + " SERIAL REFERENCES " + name + "_Dimension"
        fkeys ← fkeys ∪ fk
    end
    // Close pk string
    pk ← pk + ")"
    // Join foreign keys
    fkey_string ← fkeys.join(",\n")
    // Cast each measures into a string
    foreach member ∈ fact_table.members do
        measures ← measures ∪ string(member)
    end
    // Join strings using ",\n"
    column_string ← measures.join(",\n")
    // Create string with linebreaks
    table_statement ← "'
        CREATE TABLE {name}_Fact_Table (
        {pk},
        {fkey_string},
        {column_string}
        );"'
    ddl_tables ← ddl_tables ∪ table_statement
end
return ddl_tables
```

C. PygramETL code generation

When generating code for PygramETL the main concern is declaring dimensions and fact tables. However, for the code to be correct and functioning Python, it must include proper indentation, and import the required packages and modules. For PygramETL to compile the code there must also be a connection to the data warehouse, in addition to dimensions and fact tables. In order to maintain proper indentation, a `PythonCodeBlock` object is utilized to represent Python code. It consists of a head and a block, the head precedes the block by appearing on the line above it, and every line in the block is indented one tab (equivalent to four spaces). Although not required, dimension and fact table declarations are written over multiple lines using a `PythonCodeBlock` to increase readability of the generated Python source file. Once all dimensions and fact tables has been converted into `PythonCodeBlocks` they can be written to file by using their string representation. The algorithm for the string representation of a `PythonCodeBlock` can be seen in **Algorithm 4**. An example use of `PythonCodeBlock` is the declaration of a dimension, where the variable is declared in the head, and its name, primary key and attrributes are part of the body, this example can be seen in **Listing 6**.

```
-- DeclarativeETL
[Default]
DBMS = "postgresSQL 15.2"
pk_naming = "KEY"
schema_type = "star"
fact_measure_type = "DECIMAL(10, 4)"

[fact.Lineorder]
measures = ["quantity: INT", "price"]

-- DDL
CREATE TABLE Lineorder_Fact_Table
(
CustomerKEY SERIAL REFERENCES Customer_Dimension,
PartKEY SERIAL REFERENCES Part_Dimension,
DateKEY SERIAL REFERENCES Date_Dimension,
PRIMARY KEY (CustomerKEY, PartKEY, DateKEY),
quantity INT,
price DECIMAL(10, 4)
);
```

Listing 4. Generated DDL fact table

```
CREATE DATABASE test_dw;
\connect test_dw

CREATE TABLE Customer_Dimension
(
CustomerKEY SERIAL PRIMARY KEY,
name VARCHAR(30),
address VARCHAR(30)
);

CREATE TABLE Part_Dimension
(
PartKEY SERIAL PRIMARY KEY,
name VARCHAR(30),
manufacturer VARCHAR(30)
);

CREATE TABLE Lineorder_Fact_Table
(
CustomerKEY SERIAL REFERENCES Customer_Dimension,
PartKEY SERIAL REFERENCES Part_Dimension,
PRIMARY KEY (CustomerKEY, PartKEY),
quantity INT,
price DECIMAL(10, 4)
);
```

Listing 5. Minimal generated DDL example

Algorithm 4: `PythonCodeBlock` string method

Data: string: **Head**, list of strings: **Block**
Result: string

```
result ← Head + "\n"
foreach line ∈ Block do
    result ← result + "    " + line + "\n"
end
return result
```

The template used for generating PygramETL code is a sequence of imports, a DW connection, dimensions and fact

```
Customer_dimension = CachedDimension(    # Head
    name='Customer',                     # Block begin
    key='CustomerKEY',
    attributes=['name', 'address']) # Block end
```

Listing 6. Example of `PythonCodeBlock` string representation

table declarations. The first part of the template written to file is the statements to import PygramETL and the database driver. The latter of which depends on the DBMS and is therefore determined by the default value "DBMS", for PostgreSQL the driver is `psycopg2`. Next is to write the connection to the DW, this is very dependent on the developers setup and credentials, as such it will require tweaking after code generation, so for completeness this value is set to `dw_connection_string = "host='localhost' dbname=specification.dw_name user='user' password='pass'"`,
where `specification.dw_name` is the name specified for the DW in the specification. An example of this initial part of a generated PygramETL file can be seen in **Listing 7**.

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

# Open connection to the data warehouse.
dw_string = "host='localhost' dbname='test_dw'
    user='postgres' password='1234'"
dw_conn = psycopg2.connect(dw_string)
dw_conn_wrapper = pygrametl.ConnectionWrapper(
    connection=dw_conn)
```

Listing 7. Initial part of generated PygramETL

After the initial part, the dimensions and fact tables are now to be written. The declared tables are written following the templates seen in **Algorithm 5** and **Algorithm 6**. CachedDimension [10] is used as the default dimension type as suggested by the PygramETL documentation. Much like when generating code for DDL, the method responsible for generating PygramETL structures operates on a `IntermediateSpecification` object. For each dimension or fact table the necessary information is collected and processed into proper PygramETL formatted strings, which are then used to populate its respective string template.

For PygramETL dimensions [10], we must declare the parameters of the counterpart table in the DW, the name of the table as a string, the primary key of the table as a string, and the non-key columns (attributes) of the table as a list of strings. As seen in **Algorithm 5** only the attributes must be processed as both the name and key are already available in the `ParsedDimension`. The processing of the attributes requires extracting the name and wrapping it in single quotes ('), then appending it to a string of all attributes separated by commas (line 3), when written as PygramETL Python code it will further be wrapped in brackets to simulate the list. After processing the attributes it is as simple as populating the string template with the values (line 5). This processing and string creation can be seen in **Algorithm 5**. An example of the specification and its resulting PygramETL Python code can be seen in **Listing 8**, for the Customer dimension.

For PygramETL fact tables [10], we must also declare the parameters of its counterpart table in the DW. This means the DW fact tables' name as a string, measures as a list of strings, and foreign keys as a list of strings, denoted by "keyrefs". Both measures and foreign keys of the `ParsedFactTable` must be processed identically to that of attributes for dimensions. In order for them to be printed as a list of strings. To iterate this means extracting the names of the measure or foreign key, wrapping them in single quotes ('), separating them by commas, and finally wrapping the entire string as in brackets to simulate a list. Once all string values are prepared they are used to populate the string template (line 9). This processing and string creation can be seen in **Algorithm 6**. An example of the specification and its resulting PygramETL Python code can be seen in **Listing 9**, for the Lineorder fact table, related to the Customer dimension.

A minimal example of generated PygramETL Python code can be seen in **Listing 10**, consisting of the previous initial part, dimension and fact table.

Algorithm 5: Processing of `ParsedDimension`

Data: `ParsedDimension` dimension
Result: string of PygramETL dimension

1 $attributes \leftarrow$ """ + $dimension.members.pop()$ + """
2 **foreach** $member \in dimension.members$ **do**
3 $\quad attributes \leftarrow attributes$ + ", '" + $member.name$ + """
4 **end**
5 $table \leftarrow$''' {$dimension.name$}_Dimension = CachedDimension(
name='$dimension.name$_Dimension',
key='[{$facttable.key$}]',
attributes=[{$attributes$}])'''
6 **return** $table$

```
# DeclarativeETL
[dimension.Customer]
attributes = ["name", "address"]

# Generated Python
Customer_Dimension = CachedDimension(
    name='Customer_Dimension',
    key='CustomerKEY',
    attributes=['name', 'address'])
```

Listing 8. Generated PygramETL dimension

D. Usage

To use DeclarativeETL, the `pygrametl` python package must be installed. Once installed a developer may gain access to DeclarativeETL by importing it as follows `from pygrametl import declarativeetl`. Once imported it may be used by calling the main file generating function `create_source_files()`, which accepts

Algorithm 6: Processing of `ParsedFactTable`

Data: `ParsedFactTable` facttable
Result: string of PygramETL fact table

```
measures ← """ + facttable.members.pop
foreach member ∈ facttable.members do
    measures ← measures + ", " + member.name + """
end
keyrefs ← """ + facttable.dimensions_and_references.pop()[1] + """
foreach name, reference ∈ facttable.dimensions_and_references do
    keyrefs ← keyrefs + ", " + reference + """
end
table ←''' {facttable.name}_Fact_Table = FactTable(
              name='facttable.name',
              keyrefs=[{keyrefs}],
              measures=[{measures}])'''
return table
```

```
# DeclarativeETL
[fact.Lineorder]
measures = ["quantity: INT", "price"]

# Generated Python
Lineorder_Fact_Table = FactTable(
    name='Lineorder_Fact_Table',
    keyrefs=['CustomerKEY'],
    measures=['quantity', 'price'])
```

Listing 9. Generated PygramETL fact table

a file path to a specification as input. An example of DeclarativeETL usage can be seen in **Listing 11**. Once invoked, `create_source_files()`, will create a script source file to setup the data warehouse and Python source file, `DDL-generated-setup.ddl` and `PygramETL-generated-setup.py`, respectively, in the current working directory.

The DDL file may then be executed in a DBMS via a terminal tool, e.g. psql [11], to fully setup a database with tables as the data warehouse. After which the PygramETL Python file may be executed by invoking a Python compiler.

VI. Evaluation

DeclarativeETL is evaluated based on increase of productivity in terms of lines of code saved. DeclarativeETL is also evaluated on its performance in terms of runtime and memory usage, these are expected to be low impact, but included for the sake of completeness and to show that the increase in productivity is not hindered by long execution time. Finally, DeclarativeETL tested for its correctness, making sure the generated code can be compiled and executed.

A. Productivity

The main goal of DeclarativeETL is to improve productivity, this can be measured in the amount of lines and characters

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

# Open connection to the data warehouse.
dw_string = "host='localhost' dbname='test_dw'
    user='postgres' password='1234'"
dw_conn = psycopg2.connect(dw_string)
dw_conn_wrapper = pygrametl.ConnectionWrapper(
    connection=dw_conn)

Customer_Dimension = CachedDimension(
    name='Customer_Dimension',
    key='CustomerKEY',
    attributes=['name', 'address'])

Lineorder_Fact_Table = FactTable(
    name='Lineorder_Fact_Table',
    keyrefs=['CustomerKEY'],
    measures=['quantity', 'price'])
```

Listing 10. Minimal example of generated PygramETL Python code

```
>>> from pygrametl import declarativeetl
>>> declarativeetl.create_source_files("path\\to\\
    specification")
```

Listing 11. Example of DeclarativeETL usage

used to set up a PygramETL flow. Characters are measured in conjunction with lines, as unnecessary line breaks are often used to increase readability. The comparison will compare a DeclarativeETL specification to DDL and Python when setting up a PygramETL flow manually. The most important comparison is of the amount of lines/characters used to declare a dimension or fact table in a DeclarativeETL specification in contrast to doing so in DDL and PygramETL. For this comparison default values are not counted as they are defined once, no matter the amount of tables.

A dimension with a name, two non-key attributes, no roles, and a primary key requires 2 lines of code (52 characters) in DeclarativeETL, where as in DDL this requires 4 significant lines of code (101 characters) and also 4 lines of code (124 characters) in PygramETL Python. For such a small dimension this results in $1 - (2/(4+4)) = 1 - 0.25 = 0.75$, giving 75% fewer lines of code, in characters the difference is $1 - (52/(101+124)) = 1 - 0.23 = 0.77$, meaning 77% fewer characters. This comparison can be seen in **Listing 12**, using the Customer dimension.

The same comparison is carried out for a dimension with a name, 10 non-key attributes, and a primary key. In this case DeclarativeETL requires 2 lines of code (146 characters), albeit one of them being very long. Where as DDL and PygramETL requires 12 (240 characters) and 4 (219 characters) lines of code, respectively. Resulting in $1-(2/(12+4)) = 1-0.125 = 0.875$, 87.5% fewer lines and $1-(146/(240+219)) = 1 - 0.32 = 0.68$, 68% fewer characters, when opting to use DeclarativeETL as opposed to writing DDL and PygramETL. This comparison can be seen in **Listing 13**, using an extended Part dimension.

The comparisons show that the difference in characters

decreases (-9%) as the dimensions become bigger. However, the productivity is still increased more than 100%, when using DeclarativeETL compared to writing DDL/PygramETL.

This type of comparison is also carried out for the fact table (Lineorder) and its related dimension used in the running example, the comparison can be seen in **Listing 14**. DeclarativeETL requires 2 lines of code (53 characters) to define the fact table. Where as DDL and PygramETL requires 7 (268 characters) and 4 lines of code (249 characters), respectively. This results in $1-(2/(7+4)) = 1-0.18 = 0.82$, 82% fewer lines of code and $1-(54/(268+249)) = 1-0.10 = 0.90$, 90% fewer characters. Developer productivity will increase as more dimensions are added to the specification, as they are implicitly added to the fact table. However, that same decrease in required lines and characters will become lower, as measures are added to the specification, as they make up most of the fact table in DeclarativeETL.

```
# DeclarativeETL
[dimension.Customer]
attributes = ["name", "address"]

# DDL
CREATE TABLE Customer_Dimension
(
    CustomerKEY SERIAL PRIMARY KEY,
    name VARCHAR(30),
    address VARCHAR(30)
);

# PygramETL
Customer_Dimension = CachedDimension(
    name='Customer_Dimension',
    key='CustomerKEY',
    attributes=['name', 'address'])
```

Listing 12. Two attribute dimension in DeclarativeETL, DDL, and PygramETL

B. Runtime analysis

For runtime analysis both the execution time and the memory usage is measured. Neither is expected to be of any impact and be negligible, but for sake of completeness they are included here to show that, if anything DeclarativeETL does not require much processing power or memory, and does not incur waiting time on developers/users. The analysis is performed on a machine running the latest version of 64-Bit Windows 10 with the following hardware: CPU: i5-7200U CPU @ 2.50GHz, RAM: 16 GB, Storage: 256 GB SSD. For execution time the cProfile profiler is used. DeclarativeETL was executed 5 times for the running example and the results of the profiler were $26ms, 26ms, 26ms, 25ms, 28ms$ resulting in an average time of 26.2 ms with the shortest time being 25 ms and the longest being 28 ms.

For memory usage, memory-profiler 0.61.0 [1] is used. Memory profiling results in 22.2 MiB ($\tilde{2}$3MB) RAM used. As expected both results are so small that they is negligible.

[1]https://pypi.org/project/memory-profiler/

```
# DeclarativeETL
[dimension.Part]
attributes = ["name", "manufacturer", "category",
              "brand", "color", "type",
              "size: INT", "container",
              "retail_price: INT", "comment"]

# DDL
CREATE TABLE Part_Dimension
(
    PartKEY SERIAL PRIMARY KEY,
    name VARCHAR(30),
    manufacturer VARCHAR(30),
    category VARCHAR(30),
    brand VARCHAR(30),
    color VARCHAR(30),
    type VARCHAR(30),
    size INT,
    container VARCHAR(30),
    retail_price INT,
    comment VARCHAR(30)
);

# PygramETL
Part_Dimension = CachedDimension(
    name='Part_Dimension',
    key='PartKEY',
    attributes=['name', 'manufacturer', 'category'
                ,'brand', 'color', 'type', 'size',
                'container', 'retail_price',
                'comment'])
```

Listing 13. 10 attribute dimension in DeclarativeETL, DDL, and PygramETL

```
# DeclarativeETL
[fact.Lineorder]
measures = ["quantity: INT", "price"]

# DDL
CREATE TABLE Lineorder_Fact_Table
(
    CustomerKEY SERIAL REFERENCES
        Customer_Dimension,
    PartKEY SERIAL REFERENCES Part_Dimension,
    DateKEY SERIAL REFERENCES Date_Dimension,
    PRIMARY KEY (CustomerKEY, PartKEY, DateKEY),
    quantity INT,
    price DECIMAL(10, 4)
);

# PygramETL
Lineorder_Fact_Table = FactTable(
    name='Lineorder_Fact_Table',
    keyrefs=['CustomerKEY', 'PartKEY', 'DateKEY'],
    measures=['quantity', 'price'])
```

Listing 14. fact table in DeclarativeETL, DDL, and PygramETL

C. Correctness

The last evaluation performed is whether the generated code is correct and can be executed. The generated PygramETL Python code was executed successfully using Python 3.11 and the latest version of PygramETL. The generated DDL file was used as a script in PostgreSQL 15.2 psql. It was successfully executed and the expected database with tables was created.

VII. Conclusion and future work

This paper presents the DeclarativeETL framework, enabling simple and effective declaration of ETL for data warehousing. The user does not require any database, ETL or even programming experience. The framework is therefore very well suited for data scientists allowing them to easily set up and integrate new data warehouse solutions.

The DeclarativeETL framework facilitates the generation of PostgreSQL DDL and PygramETL Python based on a common declarative specification, for star schemas. and increases developer productivity, reducing errors and the amount of lines needed to be written. The generated code can be used as a starting point for PygramETL and the developer can extend it with data sources and transformation rules in PygramETL.

The main contributions of DeclarativeETL is to increase developer productivity by providing developers with a starting point for PygramETL and PostgreSQL, based on an easy to write specification. Evaluation shows that using DeclarativeETL is both lightweight and productive, while producing correct code.

For future work it is relevant to support the commonly used snowflake schema and slowly changing dimensions. Also relevant is it to expand the specification to support data sources to generate extraction of data, transformation rules such as requiring all data of default type to be lowercase, and more DBMS's.

References

[1] Jensen, C. S., Pedersen, T. B., & Thomsen, C. (2010). Multidimensional databases and data warehousing. Synthesis Lectures on Data Management, 2(1), 1-111. https://doi.org/10.1007/978-3-031-01841-1

[2] Thomsen, C. (2019). "ETL." In: Sakr, S., Zomaya, A.Y. (eds) Encyclopedia of Big Data Technologies. Springer, Cham. https://doi-org.zorac.aub.aau.dk/10.1007/978-3-319-77525-8_11

[3] Jensen, S.K., Thomsen, C., Pedersen, T.B., Andersen, O. (2021). "pygrametl: A Powerful Programming Framework for Easy Creation and Testing of ETL Flows." In: Hameurlain, A., Tjoa, A.M. (eds) Transactions on Large-Scale Data- and Knowledge-Centered Systems XLVIII. Lecture Notes in Computer Science(), vol 12670. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-63519-3_3

[4] Andersen, O., Thomsen, C., & Torp, K. (2018). SimpleETL: ETL Processing by Simple Specifications. In Proceedings of the 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with 10th EDBT/ICDT Joint Conference (Vol. 2062). CEUR Workshop Proceedings. CEUR Workshop Proceedings Vol. 2062 http://ceur-ws.org/Vol-2062/paper10.pdf

[5] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. "The Star Schema Benchmark and Augmented Fact Table Indexing." Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[6] Tom Preston-Werner, Pradyun Gedam, et al. "TOML: Tom's Obvious Minimal Language." https://toml.io/en/

[7] TOML v1.0.0 Documentation: Tables https://toml.io/en/v1.0.0#table

[8] Python documentation: tomllib https://docs.python.org/3/library/tomllib.html

[9] PostgreSQL 15 Documentation "Create Table" https://www.postgresql.org/docs/current/sql-createtable.html

[10] PygramETL documentation https://chrthomsen.github.io/pygrametl/doc/

[11] PostgreSQL psql https://www.postgresql.org/docs/current/app-psql.html

APPENDIX A
FULL EXAMPLE OF SCHEMA SPECIFICATION BASED ON STAR SCHEMA BENCHMARK

```
# Declare defaults
[Default]
DBMS = "postgresql 15.2"
pk_naming = "KEY"
schema_type = "star"
dimension_attribute_type = "VARCHAR(30)"
fact_measure_type = "DECIMAL(10, 4)"

----

# Further specify each dimension
[dimension.Customer]
attributes = ["name", "address", "city", "nation", "region", "phone", "mktsegment"]

[dimension.Part]
attributes = ["name", "manufacturer", "category", "brand", "color", "type", "size: INT", "container"]

[dimension.Supplier]
attributes = ["name", "address", "city", "nation", "region", "phone"]

[dimension.Date]
attributes = ["dayofweek", "month: INT", "year: INT", "yearmonthnum: INT", "yearmonth: INT", "daynumweek: INT"]
roles = ["orderdate", "commitdate"]

[fact.Lineorder]
measures = ["linenumber", "ordpriority: VARCHAR(15)", "shippriority: VARCHAR(1)", "quantity", "extendedprice", "ordtotalprice", "discount", "revenue", "supplycost", "tax", "shipmode: VARCHAR(10)"]
```

Listing 15. Full Example of schema specification

APPENDIX B
GENERATED DDL AND PYTHON CODE FOR RUNNING EXAMPLE

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

Customer_dimension = CachedDimension(
    name='Customer',
    key='CustomerKEY',
    attributes=['name', 'address'])

Part_dimension = CachedDimension(
    name='Part',
    key='PartKEY',
    attributes=['name', 'manufacturer'])

Date_dimension = CachedDimension(
    name='Date',
    key='DateKEY',
    attributes=['day', 'month', 'year'])

Lineorder_fact_table = FactTable(
    name='Lineorder',
    keyrefs=['CustomerKEY', 'PartKEY', 'DateKEY'],
    measures=['quantity', 'price'])
```

Listing 16. Generated Pygram

```
CREATE TABLE Customer (
CustomerKEY INT PRIMARY KEY,
name VARCHAR(30),
address VARCHAR(30)
);

CREATE TABLE Part (
PartKEY INT PRIMARY KEY,
name VARCHAR(30),
manufacturer VARCHAR(30)
);

CREATE TABLE Date (
DateKEY INT PRIMARY KEY,
day INT,
month INT,
year INT
);

CREATE TABLE Lineorder (
CustomerFK INT REFERENCES Customer,
PartFK INT REFERENCES Part,
DateFK INT REFERENCES Date,
quantity INT,
price DECIMAL(10, 4)
);
```

Listing 17. Generated DDL