SimpleETL: ETL Processing by Simple Specifications*

Ove Andersen
Aalborg University & FlexDanmark
Denmark
xcalibur@cs.aau.dk
oan@flexdanmark.dk

Christian Thomsen
Aalborg University
Denmark
chr@cs.aau.dk

Kristian Torp
Aalborg University
Denmark
torp@cs.aau.dk

ABSTRACT

Massive quantities of data are today collected from many sources. However, it is often labor-intensive to handle and integrate these data sources into a data warehouse. Further, the complexity is increased when specific requirements exist. One such new requirement, is the *right to be forgotten* where an organization upon request must delete all data about an individual. Another requirement is when *facts are updated* retrospectively. In this paper, we present the general framework *SimpleETL* which is currently used for Extract-Transform-Load (ETL) processing in a company with such requirements. SimpleETL automatically handles all database interactions such as creating fact tables, dimensions, and foreign keys. The framework also has features for handling version management of facts and implements four different methods for handling deleted facts. The framework enables, e.g., data scientists, to program complete and complex ETL solutions very efficiently with only few lines of code, which is demonstrated with a real-world example.

1 INTRODUCTION

Data is being collected at unprecedented speed partly due to cheaper sensor technology and inexpensive communication.

Companies have realized that detailed data is valuable because it can provide up-to-date and accurate information on how the business is doing. These changes have in recent year coined terms such as "Big Data", "The five V's", and "Data Scientist". It is, however, not enough to collect data; it should also be possible for the data scientist[1] to integrate it with existing data and to analyze it.

A data warehouse is often used for storing large quantity of data possibly integrated from many sources. A wide range of Extract-Transform-Load (ETL) tools support cleaning, structuring, and integration of data. The available ETL tools offer many advanced features, which make them very powerful but also both overwhelming and sometimes rigid in their use. It can thus be challenging for a data scientist to quickly add a new data source. Further, many of these products mainly focus on data processing and less on aspects such as database schema handling. Other important topics are privacy and anonymity concerns of citizens, which has caused the EU (and others) to introduce regulations where citizens have a *right to be forgotten* [9]. Violating these regulations can lead to large penalties and it is thus important to enable easy removal of an individual citizen's data from a data warehouse.

A simplified real-world example use case is presented by a star-schema in Figure 1, where passenger travels carried out by a taxi company are stored. Each travel is a fact stored in a fact table, connected with a vehicle, a customer, and a date dimension. It is common practice that *facts are deleted*, e.g., if it is discovered that an ordered trip two days ago was not executed anyway then the fact will be removed, or a *facts gets updates*, due to late arriving accounting information. Further, for audit reasons, it is required that changes must be tracked, e.g., if a price is updated.

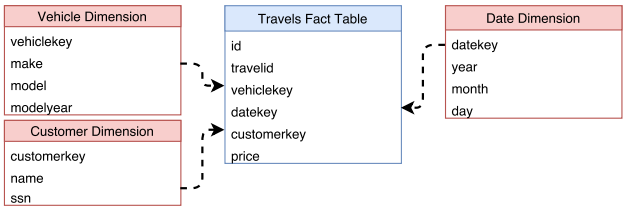

Figure 1: Example Case Star Schema

The presented SimpleETL framework enables data scientists to program an ETL solution in a very efficient and convenient way with only few lines of code mainly with specifications of metadata. The framework manages everything behind the scene from structuring data warehouse schema, fact tables, dimensions, references, indexes, and data version tracking. This also includes handling of *changes to facts* in line with Kimball's *slowly changing dimensions* [12]. Processing data using SimpleETL is automatically highly parallelized such that every dimension is handled in its own process and fact table processing is spread across multiple processes.

The rest of the paper is structured as follows: First related work is discussed in Section 2. Then a simple use-case is introduced in Section 3 followed by an example implementation in Section 4 showing how a user efficiently programs an ETL flow. In Section 5, the support for fact version management and deletion of facts is described. Then in Section 6 it is described how a data scientist configures and initializes an ETL run including how the framework operates along with a real-world use case example. Section 7 concludes the paper and points to directions for future work.

2 RELATED WORK

A survey of ETL processes and technologies is given by [16]. A plethora of ETL tools exist from commercial vendors such as IBM, Informatica, Microsoft, Oracle, and SAP [2–5, 7]. Open source ETL tools also exist such as Pentaho Data Integration and Talend [6, 8]. Gartner presents the widely used tools in its Magic Quadrant [10]. With most ETL tools, the user designs the ETL flow in a graphical user-interface by means of connecting boxes (representing transformations or operations) with arrows (representing data flows).

Another approach is taken for the tool pygrametl [14] for which it is argued that *programmatic ETL*, i.e., creating ETL programs by writing code, can be beneficial. With pygrametl, the

*Produces the permission block, and copyright information

[1]By "data scientist" we in this paper refer to someone focused at analyzing data and less in the technical aspects of DBMSs, e.g., ETL tools and Data Warehousing.

user programs Python objects for dimension and fact tables to handle insert/update operations on the target data warehouse. SimpleETL, however, hides complexity from the user and conveniently handles all schema management. Based on the specification of metadata, SimpleETL creates 1) the required SQL to generate or alter the target data warehouse schema; 2) the necessary database actions and pygrametl objects to modify the tables; and 3) processes for parallel execution. SimpleETL provides template code for its supported functionality, e.g., history tracking of changing facts. It is therefore simple and fast for a data scientist to define an ETL flow or add new sources and dimensions, because she does not have to make the code for this, but only specify the metadata.

Tomingas et al. [15] propose an approach where Apache Velocity templates and user-specified mappings are used and transformed into SQL statements. In contrast, SimpleETL is based on Python, which makes it easy for data scientists to exploit their existing knowledge and to use third party libraries.

BIAccelerator [13] is another template-based approach for creating ETL flows with Microsoft's SSIS [4], enabling properties to be defined as parameters at runtime. Business Intelligence Markup Language (Biml) [1] is a domain-specific XML-based language to define SSIS packages (as well as SQL scripts, OLAP cube definitions and more). The focus of BIAccelerator and Biml/BimlScript is to let the user define templates generating SSIS packages for repetitive tasks while SimpleETL makes it easy to create and load a data warehouse based on the templating provided by the framework.

3 USE-CASE

In this section, we describe a simplified use-case scenario that serves as a running example throughout the paper and is used to explain the distinctive features of the SimpleETL framework. The simplified use-case is heavily inspired from a real-world example.

In Figure 1, a star schema is presented, that connects information on passenger travels with a dimension for passengers, a dimension for the vehicle carrying out the travel, and a date dimension. The data is loaded from a CSV file with all the information available at each line. Both the references and measures consist of a combination of integer values, numeric values for monetary amounts, string values, date, and time values.

Every night this set of data is exported from a source system (an accounting system) and a complete data dump is available, including all historic earlier dumped data. The nightly dump has some distinctive characteristics, which make handling the data non-trivial. The characteristics are that the data contain duplicates of existing facts, contain updated measures of existing facts, and lack deleted facts, which must be detected. These three characteristics put up some special demands for the ETL solution.

Two types of requirements exist for the functionality of the final data warehouse, after data have been processed. First, a set of business-oriented demands exists, such as tracking updates of facts, e.g., when and what corrections were made. Second, updated legislation on people's rights, e.g., the General Data Protection Regulation [9], creates new requirements for data to be deleted completely if a customer requests to be forgotten.

4 FRAMEWORK COMPONENTS

This section provides an overview of the components in SimpleETL which a user customizes to create a data warehouse and corresponding ETL process. First, a class diagram is presented

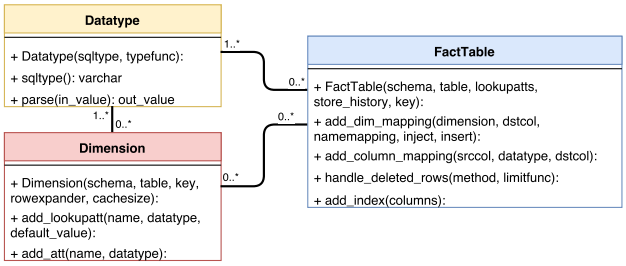

Figure 2: UML Class Diagram for SimpleETL

Listing 1: Defining Year Data Type

```
1  from simpleetl import Datatype
2  def _validate_year(val):
3  if str(val).isnumeric() and 1900 <= int(intval) <= 2099:
4  return int(intval)
5  return -1
6  yeartype = Datatype('smallint', _validate_year)
```

that shows all components in the framework. Next, each component is described in more details by from the use case in Section 3.

4.1 Class Diagram

Figure 2 shows the UML class diagram for the SimpleETL framework that consists of three classes. The class *Datatype* is used to define the type of a column in both dimension tables and (measure) in a fact table. The `parse` method transforms a value (e.g., from a string to an integer) and ensures that the data type is correct (e.g., it is a signed 32 bit integer) and any constraints on the values in the column (e.g., it must be positive). The `sqltype` method returns the SQL data type recognizable for a DBMS. The SimpleETL framework comes with most standard data types, e.g., 2, 4, and 8-byte integer, numeric, date, and string (varchar) types.

The class *Dimension* models a dimension. It is an aggregation of a number of *Datatype* objects. The *Dimension* class contains two methods, one for adding lookup attributes, `add_lookupatt`, and one for adding regular attributes, `add_att`. The combined set of lookup attributes uniquely defines a record, which refers to a *Dimension* key. Regular attributes simply describe the record. The SimpleETL framework comes with a standard date and time dimension.

The class *FactTable* models a fact table. It is an aggregation of a number of *Dimension* objects and *Datatype* objects. Four methods are available on the class, first a method for connecting a *Dimension* with the *FactTable*, `add_dim_mapping`. Second, a method for adding a measure mapping, `add_column_mapping` , a method for defining how deleted rows should be handled, `handle_deleted_rows`, and finally a method for defining additional indexes over a set of columns, `add_index`. Note that the SimpleETL framework automatically adds indexes on all dimension mappings and on the lookup attribute set.

4.2 Data Type

A data type define how a specific value is stored in the database and how a value from the data source is parsed and processed during ETL. An example of how a user can specify a data type for storing year is shown in Listing 1. The data type is defined at line 6 and named `yeartype`. The first parameter specifies the SQL data type, a 2-byte integer. The second parameter is a Python function, `_validate_year`, which both handle the diversity of data, e.g., NULL values and conversion of string representations,

Listing 2: Defining Vehicle Dimension

```
1  from simpleetl import Dimension, datatypes as dt
2  def handle_model(row, namemapping):
3  row["make"] = row["make"][0:20]
4  row["model"] = row["model"][0:20]
5  vehicledim = Dimension(schema="dims", table="vehicle",
         key="vehiclekey", rowexpander=handle_model)
6  vehicledim.add_lookupatt(name="vehicleid",
         dtype=dt.varchar(20), default_value='missing')
7  vehicledim.add_att(name="make", dtype=dt.varchar(20))
8  vehicledim.add_att(name="model", dtype=dt.varchar(20))
9  vehicledim.add_att(name="vehicleyear", dtype=yeartype)
```

and also enables constraints like 1900 <= *year* <= 2099 (line 3). If the input fails to be parsed, -1 is returned (line 5).

A number of standard data types are pre-defined, e.g., `SMALLINT` (2-byte integer), `NUMERIC(precision, scale)`, and `VARCHAR(n)`, where the length of the two latter can be defined using arguments. Floating point data types are not supported by the SimpleETL framework since it depends on equality comparison for version management and determining updates/deletes and comparing floats can yield unpredictable results. It is encouraged to use `NUMERIC(precision, scale)` when decimal values are used.

4.3 Dimension

The *Dimension* class describes how a single dimension table in the database is modeled. An example implementation of the vehicle dimension from Figure 1 is shown in Listing 2. The dimension is defined in line 5, where the first and second parameters are the schema and table name, respectively. The third parameter is the name of the primary key. The fourth parameter, `namemapping`, known from pygrametl [14], allows for a user-defined function, here `handle_model`, which is called on every row, in this case (line 2-4) truncating make and model to 20 characters, preventing overflowing the database varchar column, limited to 20 chars (line 7-8).

When the dimension has been defined, two types of attributes can be added. The first type is mandatory and is called the lookup attribute set. In the example, a vehicle id, `vehicleid`, is defined as a single lookup attribute in line 3. Lookup attributes are not allowed to be NULL as these must be comparable for lookups, hence a default value for a vehicle id is the string "missing". Adding the primary key of the *Dimension* as a single lookup attribute makes the primary key a smart key instead of a surrogate key [12]. Smart keys can optimize performance of dimension handling while a smart key can be computed, e.g., the date 2017-07-21 can be a smart key 20170721. The second set of attributes is optional and is called member attributes. Member attributes provide additional information for a dimension entry. Three member attributes are added in Listing 2 (line 7-9), adding make and model attributes as varchars of size 20 and vehicle year utilizing the `yeartype` data type, defined in Listing 1.

4.4 Fact Table

The *FactTable* class defines a fact table and all aspects of this, including database schema descriptions, data processing, and data version management. A set of lookup attributes can be defined to uniquely identifying a row. If the lookup attributes are set they enforce that duplicate facts with the same set of lookup attributes cannot exist. If no lookup attributes are defined, version management cannot be enabled and duplicate facts can exist. Lookup attributes are not allowed to have NULL values. The implementation of the fact table Travels from Figure 1 is shown in Listing 3.

Listing 3: Defining Travels Fact Table

```
1  from simpleetl import FactTable, datatypes as dt
2  travels = FactTable(schema="facts", table="travels",
       lookupatts=["travelid"], store_history=True, key="id")
3  travels.add_dim_mapping(dimension=vehicledim, dstcol="
       vehiclekey")
4  travels.add_dim_mapping(dimension=datedim, dstcol="datekey")
5  travels.add_dim_mapping(dimension=customerdim, dstcol="
       customerkey")
6  travels.add_column_mapping(srccol="id", datatype=dt.integer,
       dstcol="travelid")
7  travels.add_column_mapping(srccol="price", datatype=dt.
       numeric(6,2), dstcol="price")
8  travels.add_index(["price"])
9  travels.handle_deleted_rows(method="mark")
```

In line 2, the *FactTable* object is instantiated, given a schemma and table name as the first two parameters. The third parameter defines the lookup attributes, the fourth parameter specifies that full history should be retained and the fifth parameter defines the primary key of the table, `id`. The `lookupatts` attribute defines no two identical `travelid` can exist and is used when determining new/updated/deleted facts.

The vehicle dimension defined in Listing 2 is attached as a dimension using a single line of code in line 3. In lines 4 and 5, two additional dimensions are added, one handling date of the travel and another handling customer information, introduced in Figure 1. In line 6 and 7, two measures are added, first the lookup attribute, `id`, and second the `price` of a travel, implemented as a numeric data type.

The framework automatically creates primary keys, foreign keys, and indexes including a unique index on the lookup attributes and the primary key. It is possible for the user to add additional indexes (line 8). In line 9 it is defined that when a row is determined to have been deleted from the data source the row should be marked in the table as having been removed (method *D*4 from Section 5.2), thus keeping the fact in the data warehouse.

Overall, SimpleETL is designed to optimize productivity, ensure consistency, reduce programming errors, and help the data scientist in loading and activating data for analysis. This is realized by reuse of data types and dimensions shown using code examples and by keeping the number of methods and parameters to a minimum.

5 MODIFICATIONS OF FACTS

In some system applications it is a business requirement that facts can be updated and full history be maintained for enabling tracking of changes to facts. Simultaneously it is common practice to remove data if it is no longer valid, e.g., if a passenger travel was not carried out it is later deleted from the accounting system. Another motivation for deleting data is legal demands such as the concept called the *right to be forgotten* [9]. This section shows how these requirements are handled automatically by the SimpleETL framework.

5.1 Slowly Changing Fact Tables

To handle updates of facts we introduce the slowly changing fact table. When a user enables version tracking of facts (`store_history =True` in Listing 3 line 2), a second fact table is created.

The main fact table, illustrated in Table 1, acts a similar to a type-1 slowly changing dimension such that facts get updated (overwritten) when changes are detected in the source data. For these examples the type-1 fact table consists of a `id`, a `travelid`, shortened `tid`, and a `price`. This table is referred to as the type-1 fact table in the rest of the paper.

Table 1: T1 Facts

id	tid	price
1	100	40
2	109	25

Table 2: Version Managed Fact Table

id	tid	price	_vfrom	_vto	_ver	_fid
1	100	40	$t1$	-1	1	1
2	109	25	$t1$	-1	1	2

Table 3: Upd. T1

id	tid	price
1	100	40
2	109	35

Table 4: Updated Ver. Managed Facts

id	tid	price	_vfrom	_vto	_ver	_fid
1	100	40	$t1$	-1	1	1
2	109	25	$t1$	$t2$	1	2
3	109	35	$t2$	-1	2	2

Table 5: Del. T1 using $D2$/$D3$

id	tid	price
1	100	40

Table 6: Deleted T1 using $D4$

id	tid	price	_del
1	100	40	-1
2	109	35	$t1$

The second table, illustrated in Table 2 acts in a similar way as a type-2 version managed slowly changing dimension where version management of data is tracked using four additional columns. A pair of columns `_validfrom` and `_validto`, shortened `_vfrom` and `_vto`, stores the validity period of a fact using 32-bit Unix timestamps, $t1$ through $t3$. A version number, `_ver`, keeps track of fact changes and a column, `_fact_id`, shortened `_fid`, is references the primary key of the type-1 fact table bridging the type-1 and the version managed fact tables together, e.g., for tracing historic changes from facts in the type-1 fact table. This table is referred to as the version managed fact table in the rest of the paper.

We now illustrate what happens when a data set is loaded by the SimpleETL framework. Table 1 and Table 2 shows the type-1 and the version managed fact tables with two rows of data loaded. The `_vfrom` is set to $t1$ and the `_vto` defaults to -1 when a fact is still live. When an update happens at the data source, it is propagated to SimpleETL at the next ETL batch run. For example if the `price` for the `tid=109` is updated from 25 to 35 the measure of the type-1 fact table is overwritten, shown in Table 3, while in the version managed fact table, Table 4, the `_vto` is set for `id=2` and a new version of the fact is inserted with `id=3`.

The advantage of this two-table approach is that dispite many updates the type-1 fact table does not grow in size. The downside is increased storage cost from representing facts in both tables.

5.2 Deleting Facts

The motivation for deleting facts can be to reflect production, e.g., if a passenger travel was not carried out it is deleted in hindsight. Second, legal demands, such as the *right to be forgotten* [9], can require data to be deleted on individuals.

The SimpleETL framework enables the user to choose between four methods for handling deleting data. These are described using Table 3 and Table 4 as the outset. The fact with `tid=109` is deleted.

The first method, ***D1***, ignores when facts are deleted at the source system, i.e., if the fact with `tid=109` is deleted it will still persist in the data warehouse, like Table 3 and Table 4. This method enables keeping facts regardless of what happens at the data source and is useful if facts cannot be altered or data is loaded incrementally.

Table 7: Deleted Version Managed Facts using $D2$

id	tid	price	_vfrom	_vto	_ver	_fid
1	100	40	$t1$	-1	1	1

Table 8: Deleted Version Managed Facts using $D3$ **and** $D4$

id	tid	price	_vfrom	_vto	_ver	_fid	[$D4$ _del]
1	100	40	$t1$	-1	1	1	-1
2	109	25	$t1$	$t2$	1	2	$t3$
3	109	35	$t2$	$t3$	2	2	$t3$

The second method, ***D2***, completely deletes facts from the data warehouse if they are removed at the source system. Table 5 shows the type-1 fact table and Table 7 shows the version managed fact table after the fact with `tid=109` has been deleted. This method is useful if facts must be enforced to be removed, e.g., due to legal reasons and when data is removed at data source it will automatically be removed from the fact tables too.

The third method, ***D3***, removes the fact in the type-1 fact table, like method $D2$ shown in Table 5 while in the version managed fact table the deleted fact is marked with an time stamp `_vto=t2`, shown in Table 8. This method is useful, if the type-1 fact table must mirror the source system, while deleted data must be tracked.

The fourth method, ***D4***, adds an extra attribute to both fact tables, `_deleted`, shortened `_del`, with default value -1. When a fact is removed the `_del` measure will be set to the relevant time stamp for the fact in both the type-1 and version managed fact tables, Table 6 and Table 8 respectively. This method is useful if easy filtering of deleted facts is required for, e.g., bookkeeping on the type-1 fact table.

Having four different methods for handling deleted facts makes the SimpleETL framework very versatile and matches most business and legal needs with respect to the balance between preserving data versus privacy regulations.

6 DATA AND PROCESS FLOW

This section first introduces how the ETL process is configured and initiated, then the process flow implementation is visualized in Figure 3, separating the process flow into three stages, *Initialization* (1.1-1.4 in Figure 3), *Processing* (2.1-2.5), and *Data Migration* (3.1-3.6). White boxes in Figure 3 indicates steps processed sequentially while gray boxes indicates parallel execution.

Facts are first loaded from a data source to a data staging area and dimensional integrity is maintained with all related dimensions. Next, the data is moved from the data staging to the fact tables in three steps, first migrating updated data, then porting new data, and finally handling deleted data, according to the user specifications in Section 5. Finally a a real-world use-case is presented along with a implementation and runtime statistics.

6.1 Configuration

The SimpleETL framework supports that data is loaded from multiple data sources. Each data source is defined using a data feeder, which is a user-defined Python function that yields key/value Python dictionaries of data for every fact, e.g., one dictionary for each row in a CSV file. These dictionaries are used by the ETL process in Section 6.1. The data-feeder functions are not an integrated part of the SimpleETL framework, which allows the

Listing 4: Processing SimpleETL

```
1  prev_id = None
2  def dupfilter(row):
3  global prev_id
4  if prev_id == row["id"]:
5  return False # Ignore duplicate "id" values
6  prev_id = row["id"]
7  return True
8  def parsevehicle(row, dbcon):
9  # Split mk_mdl into two variables
10 row["make"], row["model"] = row["mk_mdl"].split("|")
11 csvfile = csv.DictReader("/path/to/file")
12 processETL(facttable=fact, datafeeder=csvfile,
       filterfunc=dupfilter, transformfunc=parsevehicle,
       [database connection details])
```

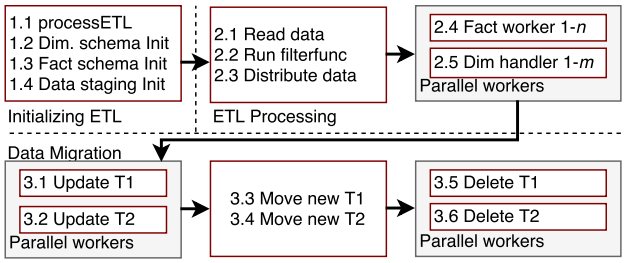

Figure 3: Main Execution Flow of SimpleETL

user to load data from various sources, e.g., CSV, ODBC, or REST APIs, only requiring that they can present a fact as a Python dictionary.

When the data warehouse structure, using the components from Section 4, and a data source are defined then the ETL process can be configured and initiated. All functionality related to database schema management and data management is handled automatically. When the ETL process has completed, the data is available in the data warehouse for querying. The ETL process is started as shown in Listing 4. In line 11, a file is prepared for loading, using Python's CSV-to-Dictionary function. The ETL process is started in line 12, where the *FactTable* and CSV file are given as input. Listing 4 also shows how two optional functions are used to customize the ETL process. The argument `filterfunc =dupfilter` defines a function for filtering rows before data is distributed to parallel workers, and the argument `processfunc =parsevehicle` defines a function distributed to all background worker processes.

We have now shown all the code that the user needs to implement in various Python function to use the SimpleETL framework. In the next section, it is described what is done internally in the framework to build the data-warehouse schema and efficiently load the data.

6.2 Initialization

Before starting the ETL data processing SimpleETL initializes database connections and validates the *FactTable* object, `processETL` (1.1) in Figure 3. Schema, constraints, and indexes are created and verified for all attached dimensions (1.2) and the fact tables (1.3). A temporary data staging table is initialized, for later handling updated and deleted facts (1.4).

6.3 Processing

The main ETL process extracts data from the data source, given the `datafeeder` argument, Figure 3 (2.1). A `filterfunc`, introduced in Section 6.1, can be applied for filtering data (2.2). Then data is distributed to the background workers (2.3) in batches of 1000 rows (user configurable size). Background fact workers (2.4) are reading and writing to the dimensions (2.5) and when all data has been processed, the fact and dimension workers commit data to the data warehouse dimensions and data staging table.

Dimension and fact handling are separated from the main process into parallel background workers of performance reasons. The background workers (2.4) and (2.5) in Figure 3, are implemented using Python's `multiprocessing.Process` and communication is handled though Inter-Process Communication (IPC) Queries. Several caching layers, using Python's `functools.lru_cache`, reduce the IPC and dimension database communication.

Parallel Fact Workers The parallel fact workers, (2.4) from Figure 3, process rows distributed in batches from (2.3). If the parameter `transformfunc` is provided, Section 6.1, this is executed first. Such a function can contain advanced user defined transformations. Second, all dimension referencing is handled using the a dimension workers (2.5). Then each measure is processed and finally the data is inserted into a data staging table. *n* parallel fact workers will be spawned where *n* equals the number of available CPU cores for the framework.

Decoupled Dimension Workers Each dimension is handled in its own separate process (2.5), i.e., having three attached dimensions will run in three separate processes. Utilizing the same dimension more than once will only spawn one instance, e.g., utilizing a date dimension three times will only use one parallel worker process. If the dimension key is a smart key, see Section 4.3, this smart key can immediately be returned from the dimension worker while surrogate keys must be co-ordinated with the dimension table, potentially with database lookups. *m* parallel dimension workers will be spawned, where *m* is the number of distinct dimensions attached a *FactTable*, see Section 4.4.

6.4 Data Migration

The data migration is split into three steps for handling updated facts, new facts, and deleted facts. The main driver, for determining updates, new data, and deleted data are the lookup attributes, see Section 4.4, which uniquely define a fact and whose values are mandatory (not NULL). Lookup attributes can be both fact measures or dimension referencing keys. If the lookup attribute set is not defined then no updating, deletion, and version management can be performed and all data will be appended.

Migrating Updated Facts Updated facts are defined as facts where the set of lookup attributes already exists in the existing fact tables and where at least one of the measures have changed. This is handled by (3.1) and (3.2) in Figure 3 and the type-1 and version managed tables are processed in parallel, as handling updates does not change relationships between these two tables.

Migrating New Facts New facts are facts whose set of lookup attributes do not exist in the type-1 and version managed fact tables. This is handled in (3.3) and (3.4) in Figure 3 where data is first migrated to the type-1 fact table and next to the version managed fact table. This sequential step is necessary as the version managed fact table needs the `id` of the type-1 fact table for referencing this. This step also ensures that no duplicate sets of lookup attributes is loaded, if the lookup attribute set of the *FactObject* is defined.

Migrating Deleted Facts If migration of deleted facts is enabled, it is determined which facts exist in the type-1 and version managed fact tables, while they do not exist in the staging table. The method for how facts are handled, when removed at the data source, is dependent on the methods described in Section 5.2.

This migration of deleted facts is handled in (3.5) and (3.6) in Figure 3.

6.5 Real-World Use

SimpleETL is designed to be a convenient and easy tool for data scientists to quickly load their data and start working with it. To show that SimpleETL also performs well a real-world use-case is implemented. One fact table is configured with version tracking enabled and deleted facts being propagated by the method *D3* from Section 5.2. The fact is constructed as *153 columns*, including 1 primary key, *41 foreign keys* to *18 dimensions*, and *111 measures*. An index is automatically generated covering the lookup attributes and the primary key and *41 indexes* are automatically generated on all the foreign keys. The data contains information on passenger travels from a fleet system. 1.2 million rows are available in a 1.67 GB CSV data file and each row has 147 columns. The final size of the type-1 and version managed fact tables are 732 and 882 MB of data and 1193 and 1422 MB of indexes, respectively.

The initial data load takes 34 minutes, including creating schema while an incremental batch providing 17 678 updated, 16 381 new, and 3 deleted facts is performed in 8 minutes on a single Ubuntu Linux server running PostgreSQL 9.6 with 16 GB of RAM, 6 core Intel Xeon E5-2695V3 CPU clocked at 2.3 GHz. The SimpleETL framework and the PostgreSQL DBMS both run on the same host.

The performance of SimpleETL scales with the number of CPUs and a large period of the execution time is related with underlying DBMS transactions. A different DBMS or configurations will yield other performance results.

7 CONCLUSION

This paper presents the SimpleETL framework that enables simple and efficient programming of ETL for data warehouse solutions without the user needs database management or ETL experience. This makes the framework particular well suited for data scientists because they can quickly integrate and explore new data sources.

The framework enables advanced fact handling such as handling slowly changing facts using version management and enables the users to decide how deleted facts should be handled. Four different methods for handling deleted facts are presented.

The framework is simple and contains only three classes for data types, dimensions, and fact tables, respectively. Each class has two to four methods. The ETL process is directed by metadata specifications and the framework handles everything else, including version management and tracking of deleted facts. The entire internal process flow extensively utilizes parallelization and IPC for processing facts and every dimension is spawned in separate processes.

The main contribution of SimpleETL is to provide a convenient and simple ETL framework for data scientists. Despite this, performance benchmarks, using real-world data scenario where facts are inserted, updated, and deleted, shows that the framework is lightweight and executing ETL batches and maintaining versioned data and deletions is performed efficiently.

There are a number of relevant directions for future work, including automatic table partitioning to handle very large data sets. Snowflake dimension support is another commonly used technique from data warehousing, which would be relevant to support in the SimpleETL framework.

REFERENCES

[1] BimlScript. http://www.bimlscript.com/. Accessed 2017-10-24.
[2] IBM InfoSphere DataStage. https://www.ibm.com/ms-en/marketplace/datastage. Accessed 2017-10-13.
[3] Informatica. https://www.informatica.com/. ([n. d.]). Accessed 2017-10-13.
[4] Microsoft SQL Server Integration Services. https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services. Accessed 2017-10-13.
[5] Oracle Data Integrator. http://www.oracle.com/technetwork/middleware/data-integrator/overview/index.html. Accessed 2017-10-13.
[6] Pentaho Data Integration - Kettle. http://kettle.pentaho.org. Accessed 2017-10-13.
[7] SAP Data Services. https://www.sap.com/products/data-services.html. Accessed 2017-10-13.
[8] Talend. https://www.talend.com/products/big-data/. Accessed 2017-10-24.
[9] 2016. EU Regulation 2016/679: General Data Protection Regulation. *Official Journal of the European Union* L119 (2016), 1–88. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC
[10] Mark A. Beyer, Eric Thoo, Mei Yang Selvage, and Ethisham Zaidi. 2017. Gartner Magic Quadrant for Data Integration Tools. (2017).
[11] Scott Curie. [n. d.]. What is Biml. http://www.bimlscript.com/walkthrough/Details/3105. Accessed 2017-10-24.
[12] Ralph Kimball and Margy Ross. 2013. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons.
[13] Reinhard Stumptner, Bernhard Freudenthaler, and Markus Krenn. 2012. *BIAccelerator – A Template-Based Approach for Rapid ETL Development.* Springer Berlin Heidelberg, 435–444.
[14] Christian Thomsen and Torben Bach Pedersen. 2009. pygrametl: a powerful programming framework for extract-transform-load programmers.. In *DOLAP*, Il-Yeol Song and Esteban ZimÃąnyi (Eds.). ACM, 49–56.
[15] Kalle Tomingas, Margus Kliimask, and Tanel Tammet. 2014. Mappings, Rules and Patterns in Template Based ETL Construction. In *The 11th International Baltic DB & IS2014 Conference*.
[16] Panos Vassiliadis. 2009. A Survey of Extract-Transform-Load Technology. 5, 1–27.