Abstract. The popularity of the Semantic Web (SW) encourages organizations to organize and publish semantic data using the RDF model. This growth poses new requirements to Business Intelligence technologies to enable On-Line Analytical Processing (OLAP)-like analysis over semantic data. The incorporation of semantic data into a Data Warehouse (DW) is not supported by the traditional Extract-Transform-Load (ETL) tools because they do not consider semantic issues in the integration process. In this paper, we propose a layer-based integration process and a set of high-level RDF-based ETL constructs required to define, map, extract, process, transform, integrate, update, and load (multidimensional) semantic data. Different to other ETL tools, we automate the ETL data flows by creating metadata at the schema level. Therefore, it relieves ETL developers from the burden of manual mapping at the ETL operation level. We create a prototype, named Semantic ETL Construct (SETLCONSTRUCT), based on the innovative ETL constructs proposed here. To evaluate SETLCONSTRUCT, we create a multidimensional semantic DW by integrating a Danish Business dataset and an EU Subsidy dataset using it and compare it with the previous programmable framework SETLPROG in terms of productivity, development time, and performance. The evaluation shows that 1) SETLCONSTRUCT uses 92% fewer Number of Typed Characters (NOTC) than SETLPROG, and SETLAUTO (the extension of SETLCONSTRUCT for generating ETL execution flows automatically) further reduces the Number of Used Concepts (NOUC) by another 25%; 2) using SETLCONSTRUCT, the development time is almost cut in half compared to SETLPROG, and is cut by another 27% using SETLAUTO; and 3) SETLCONSTRUCT is scalable and has similar performance compared to SETLPROG. We also evaluate our approach qualitatively by interviewing two ETL experts.

Keywords: Semantic ETL, Semantic Data Warehouse, RDF, Layer-based Semantic Data Integration, ETL Constructs

1. Introduction

Semantic Web (SW) technologies enable adding a semantic layer over the data; thus, the data can be processed and effectively retrieved by both humans and machines. The Linked Data (LD) principles are the set of standard rules to publish and connect data using semantic links [9]. With the growing popularity of the SW and LD, more and more organizations natively manage data using SW standards, such as Resource Description Framework (RDF), RDF-Schema (RDFs), the Web Ontology Language (OWL), etc. [17]. Moreover, one can easily convert data given in another format (database, XML, JSON, etc.) into RDF format using an RDF Wrappers [8]. As a result, a lot of semantic datasets are now available in different data portals, such as DataHub, Linked Open Data Cloud (LOD), etc. Most SW data provided by international and governmental organizations include facts and fig-
ures, which give rise to new requirements for Business Intelligence tools to enable analyses in the style of On-line Analytical Processing (OLAP) over those semantic data [34].

OLAP is a well-recognized technology to support decision making by analyzing data integrated from multiple sources. The integrated data are stored in a Data Warehouse (DW), typically structured following the Multidimensional (MD) Model that represents data in terms of facts and dimensions to enable OLAP queries. The integration process for extracting data from different sources, translating them according to the underlying semantics of the DW, and loading them into the DW is known as Extract-Transform-Load (ETL). One way to enable OLAP over semantic data is by extracting those data and translating them according to the DW’s format using a traditional ETL process. [47] outlines such a type of semi-automatic method to integrate semantic data into a traditional Relational Database Management System (RDBMS)-centric MD DW. However, the process does not maintain all the semantics of data as they are conveying in the semantic sources; hence, the integrated data no more follow the SW data principles defined in [27]. The semantics of the data in a semantic data source is defined by 1) using Internationalized Resource Identifiers (IRIs) to uniquely identify resources globally, 2) providing common terminologies, 3) semantically linking with published information, and 4) providing further knowledge (e.g., logical axioms) to allow reasoning [7].

Therefore, considering semantic issues in the integration process should be emphasized. Moreover, initiatives such as Open Government Data³ encourage organizations to publish their data using standards and non-proprietary formats [62]. The integration of semantic data into a DW raises the challenges of schema derivation, semantic heterogeneity, semantic annotation, linking as well as the schema, and data management system over traditional DW technologies and ETL tools. The main drawback of a state-of-the-art RDBMS-based DW is that it is strictly schema dependent and less flexible to evolving business requirements [16]. To cover new business requirements, every step of the development cycle needs to be updated to cope with the new requirements. This update process is time-consuming as well as costly and is sometimes not adjustable with the current setup of the DW; hence, it introduces the need for a novel approach. The limitations of traditional ETL tools to process semantic data sources are: (1) they do not fully support semantic-aware data, (2) they are entirely schema dependent (i.e., cannot handle data expressed without pre-defined schema), (3) they do not focus on meaningful semantic relationships to integrate data from disparate sources, and (4) they neither support to capture the semantics of data nor support to derive new information by active inference and reasoning on the data.

Semantic Web technologies address the problems described above, as they allow adding semantics at both data and schema level in the integration process and publish data in RDF using the LD principles. On the SW, the RDF model is used to manage and exchange data, and RDFS and OWL are used in combination with the RDF data model to define constraints that data must meet. Moreover, Data Cube (QB) [12] and Data cube for OLAP (QB4OLAP) [19] vocabularies can be used to define data with MD semantics. [44] refers to an MD DW that is semantically annotated both at the schema and data level as a Semantic DW (SDW). An SDW is based on the assumption that the schema can evolve and be extended without affecting the existing structure. Hence, it overcomes the problems triggered by the evolution of an RDBMS-based data warehousing system. On top of that, as for the physical storage of the facts and pre-aggregated values, a physically materialized SDW, the setting we focus on, store both of these as triples in the triplestore. Thus, a physical SDW is a new type of OLAP (storage) compared to classical Relational OLAP (ROLAP), Multidimensional OLAP (MOLAP), and their combination Hybrid OLAP (HOLAP) [61]. In general, physical SDW offers more expressivity at the cost of performance [35]. In [44], we proposed SETL (throughout this present paper, we call it SETL\textsubscript{PROG}), a programmable semantic ETL framework that semantically integrates both semantic and non-semantic data sources. In SETL\textsubscript{PROG}, an ETL developer has to create hand-code specific modules to deal with semantic data. Thus, there is a lack of a well-defined set of basic ETL constructs that allows developers having a higher level of abstraction and more control in creating their ETL process. In this paper, we propose a strong foundation for an RDF-based semantic integration process and a set of high-level ETL constructs that allows defining, mapping, processing, and integrating semantic data. The unique contributions of this paper are:

1. We structure the integration process into two layers: Definition Layer and Execution Layer. Dif-

³https://opengovdata.org/
different to \textit{SETLPROG} or other ETL tools, here, we propose a new paradigm: the ETL flow transformations are characterized once and for all at the Definition Layer instead of independently within each ETL operation (in the Execution Layer). This is done by generating a mapping file that gives an overall view of the integration process. This mapping file is our primary metadata source, and it will be fed (by either the ETL developer or the automatic ETL execution flow generation process) to the ETL operations, orchestrated in the ETL flow (Execution Layer), to parametrize themselves automatically. Thus, we are unifying the creation of the required metadata to automate the ETL process in the Definition layer. We propose an OWL-based Source-To-target Mapping (S2TMAP) vocabulary to express the source-to-target mappings.

2. We provide a set of high-level ETL constructs for each layer. The Definition Layer includes constructs for target schema\textsuperscript{4} definition, source schema derivation, and source-to-target mappings generation. The Execution Layer includes a set of high-level ETL operations for semantic data extraction, cleansing, joining, MD data creation, linking, inferencing, and for dimensional data update.

3. We propose an approach to automate the ETL execution flows based on metadata generated in the Definition Layer.

4. We create a prototype \textit{SETLCONSTRUCT}, based on the innovative ETL constructs proposed here. \textit{SETLCONSTRUCT} allows creating ETL flows by dragging, dropping, and connecting the ETL operations. In addition, it allows creating ETL data flows automatically (we call it \textit{SETLAUTO}).

5. We perform a comprehensive experimental evaluation by producing an MD SDW that integrates an EU farm Subsidy dataset and a Danish Business dataset. The evaluation shows that \textit{SETLCONSTRUCT} improves considerably over \textit{SETLPROG} in terms of productivity, development time, and performance. In summary: 1) \textit{SETLCONSTRUCT} uses 92\% fewer Number of Typed Characters (NOTC) than \textit{SETLPROG}, and \textit{SETLAUTO} further reduces the Number of Used Concepts (NOUC) by another 25\%; 2) using \textit{SETLCONSTRUCT}, the development time is almost cut in half compared to \textit{SETLPROG}, and is cut by another 27\% using \textit{SETLAUTO}; 3) \textit{SETLCONSTRUCT} is scalable and has similar performance compared to \textit{SETLPROG}. Additionally, we interviewed two ETL experts to evaluate our approach qualitatively.

The remainder of the paper is organized as follows. We discuss the terminologies and the notations used throughout the paper in Section 2. Section 3 explains the structure of the datasets we use as a use case. Section 4 gives the overview of an integration process. The descriptions of the Definition Layer and Execution Layer constructs are given in Sections 5 and 6, respectively. Section 7 presents the automatic ETL execution flow generation process. In Section 8, we create an MD SDW for the use case using \textit{SETLCONSTRUCT} and compare the process with \textit{SETLPROG} using different metrics. The previous research related to our study is discussed in Section 9. Finally, we conclude and give pointers to future work in Section 10.

2. Preliminary definitions

In this section, we provide the definitions of the notions and terminologies used throughout the paper.

2.1. RDF graph

An RDF graph is represented as a set of statements, called RDF triples. The three parts of a triple are subject, predicate, and object, respectively, and a triple represents a relationship between its subject and object described by its predicate. Each triple, in the RDF graph, is represented as subject $\rightarrow$ object, where subject and object of the triple are the nodes of the graph, and the label of the directed arrow corresponds to the predicate of the triple. Given $I$, $B$, and $L$ are the sets of IRIs, blank nodes, and literals, and $(I \cap B \cap L) = \emptyset$, an RDF triple is $(s, p, o)$, where $s \in (I \cup B)$, $p \in I$, and $o \in (I \cup B \cup L)$. An RDF graph $G$ is a set of RDF triples, where $G \subseteq (I \cup B) \times I \times (I \cup B \cup L)$ [26].

2.2. Semantic data source

We define a semantic data source as a Knowledge Base (KB) where data are semantically defined. A KB is composed of two components, TBox and ABox. The TBox introduces terminology, the vocab-

\textsuperscript{4}Here, we use the terms “target” and “MD SDW” interchangeably.
Logical axioms, OWL provides additional characteristics of properties. In addition to supporting RDFS constructs for property axioms, respectively [4]. Generally, a concept provides a general framework for a group of instances that have similar properties. A property either relates the instances of concepts or associates the instances of a concept to literals. Terminological axioms are used to describe the domain’s concepts, properties, and the relationships and constraints among them. In this paper, we consider a KB as an RDF graph; therefore, the components of the KB are described by a set of RDF triples. Some standard languages such as RDFS and OWL provide standard terms to define the formal semantics of a TBox. In RDFS, the core concepts rdfs:Class and rdf:Property are used to define the concepts and properties of a TBox; one can distinguish between instances and concepts by using the rdf:type property, express concept and property taxonomies by using rdfs:subClassOf and rdfs:subPropertyOf, and specify the domain and range of properties by using the rdfs:domain and rdfs:range properties. Similarly, OWL uses owl:Class to define concepts and either owl:DatatypeProperty or owl:ObjectProperty for properties. In addition to rdfs:subClassOf, it uses owl:equivalentClass and owl:disjointWith constructs for class axioms to give additional characteristics of concepts. Property axioms define additional characteristics of properties. In addition to supporting RDFS constructs for property axioms, OWL provides owl:equivalentProperty and owl:inverseOf to relate different properties, provides owl:FunctionalProperty and owl:InverseFunctionalProperty for imposing global cardinality constraints, and supports owl:SymmetricProperty and owl:TransitivityProperty for characterizing the relationship type of properties. As a KB can be defined by either language or both, we generalize the definition of C and P in a TBox T as $C(T) = \{c| \text{type}(c) \in \mathbb{P}[(\text{rdfs:Class}, \text{owl:Class})]\}$ and $P(T) = \{p| \text{type}(p) \in \mathbb{P}[(\text{rdf:Property}, \text{owl:ObjectProperty}, \text{owl:DatatypeProperty})]\}$, respectively, where type(x) returns the set of concepts of x, i.e., $(x \ \text{rdf:type} \ \text{?type}(x))$ — it returns the set of the objects of the triples whose subjects and predicates are x and rdf:type, respectively — and $\mathbb{P}(s)$ is the power set of s.

2.3. Semantic data warehouse

A semantic data warehouse (SDW) is a DW with the semantic annotations. We also considered it as a KB. Since the DW is represented with Multidimensional (MD) model for enabling On-Line Analytical Processing (OLAP) queries, the KB for an SDW needs to be defined with MD semantics. In the MD model, data are viewed in an n-dimensional space, usually known as a data cube, composed of facts (the cells of the cube) and dimensions (the axes of the cube). Therefore, it allows users to analyze data along several dimensions of interest. For example, a user can analyze sales of products according to time and store (dimensions). Facts are the interesting things or processes to be analyzed (e.g., sales of products) and the attributes of the fact are called measures (e.g., quantity, amount of sales), usually represented as numeric values. A dimension is organized into hierarchies, composed of several levels, which permit users to explore and aggregate measures at various levels of detail. For example, the location hierarchy (municipality $\rightarrow$ region $\rightarrow$ state $\rightarrow$ country) of the store dimension allows to aggregate the sales at various levels of detail. We use the QB4OLAP vocabulary to describe the multidimensional semantics over a KB [19]. QB4OLAP is used to annotate a TBox with MD components and is based on the QB vocabulary which is the W3C standard to publish MD data on the Web [15]. QB is mostly used for analyzing statistical data and does not adequately support OLAP MD constructs. Therefore, in this paper, we choose QB4OLAP. Figure 1 depicts the ontology of QB4OLAP [62]. The terms prefixed with “qb:” are from the original QB vocabulary, and QB4OLAP terms are prefixed with “qb4o:” and displayed with gray background. Capitalized terms represent OWL concepts, and non-capitalized terms represent OWL properties. Capitalized terms in italics represent concepts with no instances. The blue-colored square in the figure represents our extension of QB4OLAP ontology.

In QB4OLAP, the concept qb:DataSet is used to define a dataset of observations. The structure of the dataset is defined using the concept qb:DataStructureDefinition. The structure can be a cube (if it is defined in terms of dimensions and measures) or a cuboid (if it is defined in terms of lower levels of the dimensions and measures). The property qb4o:isCuboidOf is used to relate a cuboid to its corresponding cube. To define dimensions, levels and hierarchies, the concepts qb4o:DimensionPro-
Fig. 1. QB4OLAP vocabulary.

Property, qb4o:LevelProperty, and qb4o:Hierarchy are used. A dimension can have one or more hierarchies. The relationship between a dimension and its hierarchies is connected via the qb4o:hasHierarchy property or its inverse property qb4o:inHierarchy. Conceptually, a level may belong to different hierarchies; therefore, it may have one or more parent levels. Each parent and child pair has a cardinality constraint (e.g., 1-1, n-1, 1-n, and n-n) [62]. To allow this kind of complex nature, hierarchies in QB4OLAP are defined as a composition of pairs of levels, which are represented using the concept qb4o:HierarchyStep. Each hierarchy step (pair) is connected to its component levels using the properties qb4o:parentLevel and qb4o:childLevel. A roll-up relationship between two levels is defined by creating a property which is an instance of the concept qb4o:RollupProperty; each hierarchy step is linked to a roll-up relationship with the property qb4o:rollup and the cardinality constraint of that relationship is connected to the hierarchy step using the qb4o:pcCardinality property. A hierarchy step is attached to the hierarchies it belongs to using the property qb4o:inHierarchy [19]. The concept qb4o:LevelAttributes is used to define attributes of levels. We extend this QB4OLAP ontology (the blue-colored box in the figure) to enable different types of dimension updates (Type 1, Type 2, and Type 3) to accommodate dimension update in an SDW, which are defined by Ralph Kimball in [37]. To define the update-type of a level attribute in the TBox level, we introduce the qb4o:UpdateType class whose instances are qb4o:Type1, qb4o:Type2, and qb4o:Type3. A level attribute is connected to its update-type by the property qb4o:updateType. The level attributes are linked to its corresponding levels using the property qb4o:hasAttribute. We extend the definition of C and P of a TBox, T for an SDW as

\[
C(T) = \{ c | type(c) \in \{\text{rdfs:Class}, \text{owl:Class}, \text{qb:DataStructureDefinition}, \text{qb:DataSet}, \text{qb:DimensionProperty}, \text{qb4o:LevelProperty}, \text{qb4o:Hierarchy}, \text{qb4o:HierarchyStep}\} \}
\]
\[ P(T) = \{ p | \text{type}(p) \in \mathbb{P}((\text{rdf:Property, owl:ObjectProperty, owl:Datatype Property, qb4o:LevelAttribute, qb:MeasureProperty, qb4o:Rollup Property})) \} \] (2)

3. A use case

We create a semantic Data Warehouse (SDW) by integrating two data sources, namely, a Danish Agriculture and Business knowledge base and an EU Farm Subsidy dataset. Both data sources are described below.

Description of Danish Agriculture and Business knowledge base. The Danish Agriculture and Business knowledge base integrates a Danish Agricultural dataset and a Danish Business dataset. The knowledge base can be queried through the SPARQL endpoint [http://extbi.lab.aau.dk:8080/sparql/](http://extbi.lab.aau.dk:8080/sparql/). In our use case, we only use the business related information from this knowledge base and call it the Danish Business dataset (DBD). The relevant portion of the ontology of the knowledge base is illustrated in Fig. 2. Generally, in an ontology, a concept provides a general description of the properties and behavior for the similar type of resources; an object property relates among the instances of concepts; a data type property is used to associate the instances of a concept to literals.

We start the description from the concept bus: Owner. This concept contains information about the owners of companies, the type of the ownership, and the start date of the company ownership. A company is connected to its owner through the bus:hasOwner property. The bus:Company concept is related to bus:BusinessFormat and bus:ProductionUnit through the bus:hasProductionUnit and bus:hasFormat properties. Companies and their production units have one or more main and secondary activities. Each company and each production unit has a postal address and an official address. Each address is

![Fig. 2. The ontology of the Danish Business dataset. Due to the large number of datatype properties, they are not included.](http://extbi.lab.aau.dk:8080/sparql/)
positioned at an address feature, which is in turn contained within a particular municipality.

**Description of the EU subsidy dataset** Every year, the European Union provides subsidies to the farms of its member countries. We collect EU Farm subsidies for Denmark from https://data.farmsubsidy.org/Old/. The dataset contains two MS Access database tables: Recipient and Payment. The Recipient table contains the information of recipients who receive the subsidies, and the Payment table contains the amount of subsidy given to the recipients. We create a semantic version of the dataset using SETLPROG of subsidy given to the recipients. We create a schema of datatype properties, all are not included.

Fig. 3. The ontology of the Subsidy dataset. Due to the large number of its member countries. We collect EU Farm subsidies for Denmark from https://data.farmsubsidy.org/Old/. The dataset contains two MS Access database tables: Recipient and Payment. The Recipient table contains the information of recipients who receive the subsidies, and the Payment table contains the amount of subsidy given to the recipients. We create a semantic version of the dataset using SETLPROG framework [43]. We call it the Subsidy dataset. At first, we manually define an ontology, to describe the schema of the dataset, and the mappings between the ontology and database tables. Then, we populate the ontology with the instances of the source database files. Figure 3 shows the ontology of the Subsidy dataset.

**Example 1.** Listing 1 shows the example instances of bus:Company from the Danish Business dataset and sub:Recipient and sub:Subsidy from the EU Subsidy dataset.

**Description of the Semantic Data Warehouse** Our goal is to develop an MD Semantic Data Warehouse (SDW) by integrating the Subsidy and the DBD datasets. The sub:Recipient concept in the Subsidy dataset contains the information of recipient id, name, address, etc. From bus:Company in the DBD, we can extract information of an owner of a company who received the EU farm subsidies. Therefore, we can integrate both DBD and Subsidy datasets. The ontology of the MD SDW to store EU subsidy information corresponding to the Danish companies is shown in Fig. 4, where the concept sdw:Subsidy represents the facts of the SDW. The SDW has two dimensions, namely sdw:Benificiary and sdw:Time. The dimensions are shown by a box with dotted-line in Fig. 4. Here, each level of the dimensions are represented by a concept, and the connections among levels are represented through object properties.

Listing 1. Example instances of the DBD and the Subsidy dataset are semantically defined and the goal is to develop an SDW. The first step of building an SDW is to annotate them with the QB4OLAP vocabulary. In the former, the SDW’s TBox is obtained through object properties.

## Example of sub:Recipient instances.

recipient:291894 rdf:type sub:Recipient;
sub:recipientID 291894;
sub:name "Anders Kristian Kristensen";
sub:address "Donskaervej 31,Vemb";
sub:zipcode 7570.

## Examples of bus:Company instances

company:10058996 rdf:type bus:Company;
bus:companyID 10058996;
bus:name "Regerupgard v/Kim Jonni Larsen";
bus:mainActivity activity:11100;
bus:secondaryActivity activity:682040;
bus:hasFormat businessType:Enkeltmandsvirksomhed;
bus:hasOwner owner:4000170495_10165164;
bus:name "Idomlund 1 Vindmollelaug I/S";
bus:mainActivity activity:351100;
bus:hasFormat businessType:Interessentskab;
bus:hasOwner owner:4000175029_10058996;
bus:name "Idomlund 1 Vindmollelaug I/S";
bus:mainActivity activity:351100;
bus:hasFormat businessType:Enkeltmandsvirksomhed;
bus:hasOwner owner:4000170495_10165164;
bus:name "Idomlund 1 Vindmollelaug I/S";
bus:mainActivity activity:351100;

## Example of sub:Subsidy instances.

recipient:291894 rdf:type sub:Recipient;
sub:recipientID 291894;
sub:name "Anders Kristian Kristensen";
sub:address "Donskaervej 31,Vemb";
sub:zipcode 7570.

## Example of sub:Recipient instances.

recipient:291894 rdf:type sub:Recipient;
sub:recipientID 291894;
sub:name "Anders Kristian Kristensen";
sub:address "Donskaervej 31,Vemb";
sub:zipcode 7570.

### Business Dataset

1. PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2. PREFIX bus: <http://extbi.lab.aau.dk/ontology/business/>
3. PREFIX company: <http://extbi.lab.aau.dk/ontology/business/>
4. PREFIX activity: <http://extbi.lab.aau.dk/ontology/activity/>
5. PREFIX businessType: <http://extbi.lab.aau.dk/ontology/businessType/>
6. PREFIX owner: <http://extbi.lab.aau.dk/ontology/business/Owner#>
7. # Examples of bus:Company instances
8. company:10058996 rdf:type bus:Company;
9. bus:companyID 10058996;
10. bus:name "Regerupgard v/Kim Jonni Larsen";
11. bus:mainActivity activity:11100;
12. bus:secondaryActivity activity:682040;
13. bus:hasFormat businessType:Enkeltmandsvirksomhed;
14. bus:hasOwner owner:4000170495_10165164;
15. bus:name "Idomlund 1 Vindmollelaug I/S";
16. bus:mainActivity activity:351100;
17. bus:hasFormat businessType:Interessentskab;
18. bus:hasOwner owner:4000175029_10058996;
19. bus:name "Idomlund 1 Vindmollelaug I/S";
20. bus:mainActivity activity:351100;
21. bus:companyId 10165164;
22. bus:officialaddress "Donskaervej 31,Vemb".

### Subsidy Dataset

1. PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2. PREFIX sub: <http://extbi.lab.aau.dk/ontology/subsidy/>
3. PREFIX recipient: <http://extbi.lab.aau.dk/ontology/subsidy/Recipient#>
4. PREFIX subsidy: <http://extbi.lab.aau.dk/ontology/subsidy/Subsidy#>
5. # Example of sub:Recipient instances
6. recipient:291894 rdf:type sub:Recipient;
7. sub:recipientID 291894;
8. sub:name "Anders Kristian Kristensen";
9. sub:address "Donskaervej 31,Vemb";
10. sub:municipality "Holstebro";
11. sub:zipcode 7570.

# Overview of the integration process

In this paper, we assume that all given data sources are semantically defined and the goal is to develop an SDW. The first step of building an SDW is to design its TBox. There are two approaches to design the TBox of an SDW, namely source-driven and demand-driven [61]. In the former, the SDW’s TBox is obtained by analyzing the sources. Here, ontology alignment techniques [40] can be used to semi-automatically define the SDW. Then, designers can identify the multi-dimensional constructs from the integrated TBox and annotate them with the QB4OLAP vocabulary. In the
latter, SDW designers first identify and analyze the needs of business users as well as decision makers, and based on those requirements, they define the target TBox with multidimensional semantics using the QB4OLAP vocabulary. How to design a Target TBox is orthogonal to our approach. Here, we merely provide an interface to facilitate creating it regardless of whatever approach was used to design it.

After creating the TBox of the SDW, the next step is to create the ETL process. ETL is the backbone process by which data are entered into the SDW and the main focus of this paper. The ETL process is composed of three phases: extraction, transformation, and load. A phase is a sub-process of the ETL which provides a meaningful output that can be fed to the next phase as an input. Each phase includes a set of operations. The extraction operations extract data from the data sources and make it available for further processing as intermediate results. The transformation operations are applied on intermediate results, while the load operations load the transformed data into the DW. The intermediate results can be either materialized in a data staging area or kept in memory. A data staging area (temporary) persists data for cleansing, transforming, and future use. It may also prevent the loss of extracted or transformed data in case of the failure of the loading process.

As we want to separate the metadata needed to create ETL flows from their execution, we introduce a two-layered integration process, see Fig. 5. In the Definition Layer, a single source of metadata truth is defined. This includes: the target SDW, semantic representation of the source schemas, and a source to target mapping file. Relevantly, the metadata created represents the ETL flow at the schema level. In the Execution Layer, ETL data flows based on high-level operations are created. This layer executes the ETL flows for instances (i.e., at the data level). Importantly, each ETL operation is fed the metadata created to parameterize themselves automatically. Additionally, the Execution Layer automatically checks the correctness of the created flow, by checking the compatibility of the output and input of consecutive operators. Overall the data integration process requires the following four steps in the detailed order.
1. Defining the target TBox with MD semantics using QB and QB4OLAP constructs. In addition, the TBox can be enriched with RDFS/OWL concepts and properties. However, we do not validate the correctness of the added semantics beyond the MD model. This step is done at the Definition Layer.

2. Extracting source TBoxes from the given sources. This step is done at the Definition Layer.

3. Creating mappings among source and target constructs to characterize ETL flows. The created mappings are expressed using the proposed S2TMAP vocabulary. This step is also done at the Definition Layer.

4. Populating the ABox of the SDW implementing ETL flows. This step is done at the Execution Layer.

Figure 5 illustrates the whole integration process and how the constructs of each layer communicate with each other. Here, we introduce two types of constructs: tasks and operations. On the one hand, a task requires developer interactions with the interface of the system to produce an output. Intuitively, one may consider the tasks output as the required metadata to automate operations. On the other hand, from the given metadata, an operation produces an output. The Definition Layer consists of two tasks (TargetTBoxDefinition and SourceToTargetMapping) and one operation (TBoxExtraction). These two tasks respectively address the first and third steps of the integration process mentioned above, while the TBoxExtraction operation addresses the second step. This is the only operation shared by both layers (see the input of SourceToTargetMapping in Fig. 5). Therefore, the Definition Layer creates three types of metadata: target TBox (created by TargetTBoxDefinition), source TBoxes (created by TBoxExtraction), and source-to-target mappings (created by SourceToTargetMapping). The Execution Layer covers the fourth step of the integration process and includes a set of operations to create data flows from the sources to the target. Figure 5 shows constructs (i.e., the Mediator Constructs) used by the ETL task/operations to communicate between them. These mediatory constructs store the required metadata created to automate the process. In the figure, ExtOp, TransOp, and LoadOp are the set of extraction, transformation, and load operations used to create the flows at the instance level. The ETL data flows created in the Execution Layer are automatically validated by checking the compatibility of the operations. Precisely, if an operation $O_i$’s ou-
Table 1
Summary of the ETL operations

<table>
<thead>
<tr>
<th>Operation category</th>
<th>Operation name</th>
<th>Compatible successors</th>
<th>Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extraction</td>
<td>GraphExtractor</td>
<td>GraphExtractor,</td>
<td>It retrieves an RDF graph in terms of RDF triples from semantic data sources.</td>
</tr>
<tr>
<td>Transformation</td>
<td>TBoxExtraction</td>
<td>TransformationOnLiteral,</td>
<td></td>
</tr>
<tr>
<td>TransformationOnLiteral</td>
<td>JoinTransformation, LevelMemberGenerator, ObservationGenerator, Loader</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JoinTransformation</td>
<td>TransformationOnLiteral, JoinTransformation, LevelMemberGenerator, ObservationGenerator, Loader</td>
<td>It joins two data sources and transforms the data according to the expressions described in the source-to-target mapping.</td>
<td></td>
</tr>
<tr>
<td>LevelMemberGenerator (QB4OLAP construct)</td>
<td>Loader</td>
<td>It populates levels of the target with the input data.</td>
<td></td>
</tr>
<tr>
<td>ObservationGenerator (QB4OLAP construct)</td>
<td>Loader</td>
<td>It populates facts of the target with the input data.</td>
<td></td>
</tr>
<tr>
<td>DataChangeDetector</td>
<td>LevelMemberGenerator, UpdateLevel</td>
<td>It returns the differences between the new source dataset and the old one.</td>
<td></td>
</tr>
<tr>
<td>UpdateLevel</td>
<td>Loader</td>
<td>It reflects the changes occurred in the source data to the target level.</td>
<td></td>
</tr>
<tr>
<td>MaterializeInference</td>
<td>Loader</td>
<td>It enriches the SDW by materializing the inferred triples.</td>
<td></td>
</tr>
<tr>
<td>Externallinking</td>
<td>Loader</td>
<td>It links internal resources with external KBs.</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Loader</td>
<td>It loads the data into the SDW.</td>
<td></td>
</tr>
</tbody>
</table>

put is accepted by $O_2$, then we say $O_2$ is compatible with $O_1$ and express it as $O_1 \rightarrow O_2$.

Since traditional ETL tools (e.g., PDI) do not have ETL operations supporting the creation of an SDW, we propose a set of ETL operations for each phase of the ETL to process semantic data sources. The operations are categorized based on their functionality. Table 1 summarizes each operation with its corresponding category name, compatible successors, and its objectives. Next, we present the details of each construct of the layers presented in Fig. 5.

5. The definition layer

This layer contains two tasks (TargetTBoxDefinition and Source2TargetMapping) and one operation TBoxExtraction. The semantics of the tasks are described below.

TargetTBoxDefinition The objective of this task is to define a target TBox with MD semantics. There are two main components of the MD structure: dimensions and cubes. To formally define the schema of these components, we use the notation from [12] with some modifications.

Definition 1. A dimension schema can formally be defined as a 5-tuple $(D_{name}, \mathcal{L}, \rightarrow, \mathcal{H}, \mathcal{F}_R)$ where

- $D_{name}$ is the name of the dimension;
- $\mathcal{L}$ is a set of level tuples $(L_{name}, L_A)$ such that $L_{name}$ is the name of a level and $L_A$ is the set of attributes describing the level $L_{name}$. There is a unique bottom level (the finest granularity level)
$L_b$, and unique top level (the coarsest one) denoted $L_{All}$, such that $(L_{All}, \emptyset) \in \mathcal{L}$;

→ is a strict partial order on $\mathcal{L}$. The poset $(\mathcal{L}, \rightarrow)$ can be represented as a directed graph where each node represents an aggregation level $L \in \mathcal{L}$, and every pair of levels $(L_i, L_j)$ in the poset are represented by a directed edge, also called the roll-up (i.e., aggregation) relationship, from the finer granularity level $L_i$ to the coarser granularity level $L_j$, which means that $L_i$ rolls up to $L_j$ or $L_j$ rolls down to $L_i$. Each distinct path between the $L_b$ and $L_{All}$ is called a hierarchy;

$\mathcal{H}$ is a set of hierarchy tuples $(H_{\text{name}}, H_{\text{lb}})$ where $H_{\text{name}}$ is the name of a hierarchy and $H_{\text{lb}} \subseteq \mathcal{L}$ is the set of levels composing the hierarchy. The directed subgraph formed by this set of levels must be connected; and

$\text{RUP}_{\text{Li}}^{\text{Lj}}$ is the property used to relate instances based on the roll-up relationship between $L_i$ and $L_j$ within an hierarchy. $\mathcal{F}_R$ denotes the set of all RUPs.

**Example 2.** Figure 4 shows that our use case MD SDW has two dimensions: sdw:Time and sdw:Beneficiary. The dimension schema of sdw:Time is formally defined as follows:

1. $D_{\text{name}} = \text{sdw:Time}$;
2. $\mathcal{L} = \{(\text{sdw:Day}, (\text{sdw:dayId}, \text{sdw:dayName})), (\text{sdw:Month}, (\text{sdw:monthId}, \text{sdw:monthName})), (\text{sdw:Year}, (\text{sdw:yearId}, \text{sdw:yearName}))\}$;
3. $(\rightarrow) = \{(\text{sdw:Day}, \text{sdw:Month}), (\text{sdw:Month}, \text{sdw:Year}), (\text{sdw:Year}, \text{sdw:All})\}$;
4. $\mathcal{H} = \{\text{sdw:TimeHierarchy}, \{\text{sdw:Day}, \text{sdw:Month}, \text{sdw:Year}, \text{sdw:All}\}\}$; and
5. $\mathcal{F}_R = \{\text{RUP}_{\text{sdw:Month}}^{\text{sdw:Day}} = \text{sdw:payMonth}, \text{RUP}_{\text{sdw:Year}}^{\text{sdw:Month}} = \text{sdw:payYear}, \text{RUP}_{\text{sdw:All}}^{\text{sdw:Year}} = \text{sdw:payAll}\}$.

**Definition 2.** A cube schema is a 4-tuple $(C_{\text{name}}, D_b, M, \mathcal{F}_A)$, where

- $C_{\text{name}}$ is the name of the cube;
- $D_b$ is a finite set of bottom levels of dimensions, with $|D_b| = n$, corresponding to $n$ bottom levels of $n$ dimension schemas different from each other;
- $M$ is a finite set of attributes called measures, and each measure $m \in M$ has an associated domain $\text{Dom}(m)$; and
- $\mathcal{F}_A$ is a mathematical relation that relates each measure to one or more aggregate function in $A = \{\text{SUM, MAX, AVG, MIN, COUNT} \ldots\}$, i.e., $\mathcal{F}_A \subseteq M \times A$.

**Example 3.** The cube schema of our use case, shown in Fig. 4, is formally defined as follows:

1. $C_{\text{name}} = \text{sdw:Subsidy}$;
2. $D_b = \{\text{sdw:Day, sdw:Recipient}\}$;
3. $M = \{\text{sdw:amountEuro}\}$; and
4. $\mathcal{F}_A = \{(\text{sdw:amountEuro, SUM}), (\text{sdw:amountEuro, AVG})\}$.

In Section 2.3, we discussed how the QB4OLAP vocabulary is used to define different constructs of an SDW. Listing 2 represents the sdw:Time dimension and sdw:Subsidy cube in QB4OLAP.

**TBoxExtraction** After defining a target TBox, the next step is to extract source TBoxes. Typically, in a semantic source, the TBox and ABox of the source are provided. Therefore, no external extraction task/operation is required. However, sometimes, the source contains only the ABox, no TBox. In that scenario, an extraction process is required to derive a TBox from the ABox. Since the schema level mappings are necessary to create the ETL process, and the ETL process will extract data from the ABox, we only consider the intentional knowledge available in the ABox in the TBox extraction process. We formally define the process as follows.

**Definition 3.** The TBox extraction operation from a given ABox, $\text{ABox}$ is defined as $f_{\text{ABox2TBox}}(\text{ABox}) \rightarrow \text{TBox}$. The derived TBox is defined in terms of the following TBox constructs: a set of concepts $C$, a set of concept taxonomies $H$, a set of properties $P$, and the sets of property domains $D$ and ranges $R$. The following steps describe the process to derive each TBox element for TBox.

1. $C$: By checking the unique objects of the triples in $\text{ABox}$ where $\text{rdf:type}$ is used as a predicate, $C$ is identified.
2. $H$: The taxonomies among concepts are identified by checking the instances they share among themselves. Let $C_1$ and $C_2$ be two concepts. One of the following taxonomic relationships holds between them: 1) if $C_1$ contains all instances of $C_2$, then we say $C_2$ is a subclass of $C_1 (C_2 \text{ rdfs:subClassOf } C_1)$; 2) if they do not share any instances, they are disjoint ($C_1 \text{ owl:disjointWith } C_2$); and 3) if $C_1$ and $C_2$
Listing 2. QB4OLAP representation of sdw:Time dimension and sdw:Subsidy cube

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>sdw</td>
<td><a href="http://exml.lab.sfu.ca/ontology/sdw/">http://exml.lab.sfu.ca/ontology/sdw/</a></td>
</tr>
<tr>
<td>rdf</td>
<td><a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#">http://www.w3.org/1999/02/22-rdf-syntax-ns#</a></td>
</tr>
<tr>
<td>rdfs</td>
<td><a href="http://www.w3.org/2000/01/rdf-schema#">http://www.w3.org/2000/01/rdf-schema#</a></td>
</tr>
<tr>
<td>qb</td>
<td><a href="http://purl.org/linked-data/cube#">http://purl.org/linked-data/cube#</a></td>
</tr>
</tbody>
</table>

### Time Dimension
- sdw:Time rdf:type qb:DimensionProperty;
- rdfs:label "Time Dimension";
- qb4o:hasHierarchy sdw:TimeHierarchy.

### Dimension Hierarchies
- sdw:TimeHierarchy rdf:type qb4o:Hierarchy;
- rdfs:label "Time Hierarchy";
- qb4o:inHierarchy sdw:Time.
- qb4o:hasLevel sdw:Day, sdw:Month, sdw:Year.

### Level attributes
- sdw:Day rdf:type sdw:LevelAttribute;
- rdfs:label "Day Level";
- qb4o:hasAttribute sdw:dayId, sdw:dayName.
- sdw:Month rdf:type sdw:LevelProperty;
- rdfs:label "Month Level";
- qb4o:hasAttribute sdw:monthId, sdw:monthName.
- sdw:Year rdf:type sdw:LevelProperty;
- rdfs:label "Year Level";
- qb4o:hasAttribute sdw:yearId, sdw:yearName.
- sdw:All rdf:type sdw:LevelProperty;
- rdfs:label "All".

### Level attributes
- sdw:dayId rdf:type qb4o:LevelAttribute;
- rdfs:label "day ID";
- qb4o:updateType qb4o:Type2;
- rdfs:range xsd:Integer.
- sdw:monthId rdf:type qb4o:LevelAttribute;
- rdfs:label "month ID";
- qb4o:updateType qb4o:Type2;
- rdfs:range xsd:Integer.
- sdw:yearId rdf:type qb4o:LevelAttribute;
- rdfs:label "year ID";
- qb4o:updateType qb4o:Type2;
- rdfs:range xsd:Integer.

### Rollup relations
- sdw:payMonth rdf:type qb4o:RollupProperty;
- rdfs:label "pay Month";
- qb4o:hasHierarchy sdw:Month.
- sdw:payYear rdf:type qb4o:RollupProperty;
- rdfs:label "pay Year";
- qb4o:hasHierarchy sdw:Year.
- sdw:payAll rdf:type qb4o:RollupProperty;
- rdfs:label "pay All".

### Hierarchy Steps
- _ht2 rdf:type qb4o:HierarchyStep;
- qb4o:inHierarchy sdw:TimeHierarchy;
- qb4o:childLevel sdw:Day.
- qb4o:parentLevel sdw:Month.
- _ht3 rdf:type qb4o:HierarchyStep;
- qb4o:inHierarchy sdw:TimeHierarchy;
- qb4o:childLevel sdw:Month.
- qb4o:parentLevel sdw:Year.
- _ht4 rdf:type qb4o:HierarchyStep;
- qb4o:inHierarchy sdw:TimeHierarchy;
- qb4o:childLevel sdw:Year.
- qb4o:parentLevel sdw:All.

### Subsidy Cube
- sdw:amountEuro rdf:type qb:MeasureProperty;
- qb4o:hasAttribute sdw:amountEuro;
- sdw:SubsidyStructure rdf:type qb:DataStructureDefinition;
- qb4o:component sdw:level sdw:amount;
- qb:component qb4o:level sdw:Day;
- qb:component qb4o:measure sdw:amount;
- qb4o:aggregateFunction qb4o:sum, qb4o:avg.

The above representation describes how to transform the original data into a semantic data warehouse. It uses the QB4OLAP framework to define the dimension and fact hierarchies, as well as the rollup relationships. The Time dimension is broken down into Day, Month, and Year levels, while the Subsidy dataset is structured with a measure property named amountEuro.

---

3. P, D, R: By checking the unique predicates of the triples, P is derived. A property p ∈ P can relate resources with either resources or literals. If the objects of the triples where p is used as predicates are IRIs, then p is an object property; the domain of p is the set of the types of the subjects of those triples, and the range of p is the types of the objects of those triples. If the objects of the triples where p is used as predicates are literals, then p is a datetype property; the domain of p is the set of types the subjects of those triples, and the range is the set of data types of the literals.

Note that proving the formal correctness of the approach is beyond the scope of this paper and left for future work.

**SourceToTargetMapping** Once the target and source TBoxes are defined, the next task is to characterize the ETL flows at the Definition Layer by creating source-to-target mappings. Because of the heterogeneous nature of source data, mappings among sources and the target should be done at the TBox level. In principle, mappings are constructed between sources and the target; however, since mappings can get very complicated, we allow to create a sequence of SourceToTargetMapping definitions whose subsequent input is generated by the preceding operation. The communication between these operations is by means of a materialized intermediate mapping definition and it is meant to facilitate the creation of very complex flows (i.e., mappings) between source and target.

A source-to-target mapping is constructed between a source and a target TBox, and it consists of a set of concept-mappings. A concept-mapping defines i) a relationship (equivalence, subsumption, supersumption, or join) between a source and the corresponding target concept, ii) which source instances are mapped (either all or a subset defined by a filter condition), iii) the rule to create the IRIs for target concept instances, iv) the source and target ABox locations, v) the common properties between two concepts if their relationship is join, vi) the sequence of ETL operations required to process the concept-mapping, and vii) a set of property-mappings for the properties having the target concept as a domain. A property-mapping defines how a target property is mapped from either a source property or an expression over properties.
Definition 4. Let \( T_S \) and \( T_T \) be a source TBox and a target TBox. We formally define a source-to-target mapping as a set of concept-mappings, wherein each concept-mapping is defined with a 10-tuple formalizing the elements discussed above (i–vii):

\[
\text{SourceToTargetMapping}(T_S, T_T) = \{(c_s, \text{relation}, c_t, \text{loc}_{c_s}, \text{loc}_{c_t}, \text{mapIns}, p_{\text{map}}, \text{tiniri}, p_{\text{com}}, \text{op})\}.
\]

The semantics of each concept-mapping tuple is given below.

- \( c_s \in C(T_S) \) and \( c_t \in C(T_T) \) are a source and a target concept respectively, where \( C(T) \) defined in Equation (1).
- \( \text{relation} \in \{=, \leq, \geq, >, <, \geq, \leq, \neq\} \) represents the relationship between the source and target concept. The relationship can be either equivalence (\( c_s \equiv c_t \)), supersumption (\( c_s \supseteq c_t \)), subsumption (\( c_s \subseteq c_t \)), or join. A join relationship can be either a natural join (\( c_s \bowtie c_t \)), a right-outer join (\( c_s \bowti c_t \)), or a left-outer join (\( c_s \bowtie c_t \)). A join relationship exists between two sources when there is a need to populate a target element (a level, a (QB) dataset, or a concept) from multiple sources. Since a concept-mapping represents a binary relationship, to join \( n \) sources, an ETL process requires \( n - 1 \) join concept-mappings. A concept-mapping with a join relationship requires two sources (i.e., the concept-mapping source and target concepts) as input and updates the target concept according to the join result. Thus, for multi-way joins, the output of a concept-mapping is a source concept of the next concept-mapping.
- \( \text{loc}_{c_s} \) and \( \text{loc}_{c_t} \) are the locations of source and target concept ABoxes.
- \( \text{mapIns} \in \{\text{[All]} \cup \text{FilterCondition}\} \) indicates which instances of the source concept to use to populate the target concept; it can either be all source instances or a subset of source instances defined by a filter condition.
- \( p_{\text{map}} = \{(p_{c_s}, p_{c_t})\} \) is a set of property-mappings across the properties of \( c_s \) and \( c_t \). \( p_{c_s} \) can be a property from \( \text{property}(c_s) \) or an expression over the elements of \( \text{exp}((\text{property}(c_s)) \cup \text{property}(c_t)) \) and \( p_{c_t} \) is a property from \( \text{property}(c_t) \). Here, \( \text{property}(c) \) returns the union of the set of properties which are connected with concept \( c \) either using the \text{rdfs:domain} or \text{qb4olap:inLevel} properties, or the set of roll-up properties related to \( c \). An expression allows to apply arithmetic operations and/or some high-level functions for manipulating strings, data types, numbers, dates defined as standard SPARQL functions in [25] over the properties.
- \( \text{tiniri} \) indicates how the unique IRIs of target instances are generated. The IRIs can be either the same as the source instances, or created using a property of \( c_s \), or using an expression from \( \text{exp}(\text{property}(c_s)) \), or in an incremental way.
- \( p_{\text{com}} = \{(s\text{com}, t\text{com}) | s\text{com} \in \text{property}(c_s), t\text{com} \in \text{property}(c_t)\} \) is a set of common property pairs. In each pair, the first element is a source property and the second one is a target property. \( p_{\text{com}} \) is required when the relationship between the source and target concept is a join.
- \( \text{op} \) is an ETL operation or a sequence of ETL operations (mentioned in Table 1) required to implement the mapping element in the ABox level. When \( \text{op} \) is a sequence of ETL operations, the location of the input ABox location for the first operation in the sequence is \( \text{loc}_{c_s} \); the subsequent operations in the sequence take the output of their preceding operation as the input ABox. This generation of intermediate results is automatically handled by the automatic ETL generation process described in Section 7.

In principle, an SDW is populated from multiple sources, and a source-to-target ETL flow requires more than one intermediate concept-mapping definitions. Therefore, a complete ETL process requires a set of source-to-target mappings. We say a mapping file is a set of source-to-target mappings. Definition 5 formally defines a mapping file.

Definition 5.

Mapping file = \( \bigcup_{i \in S} \text{SourceToTargetMapping}(T_i, T_j) \),

where \( S \) is the set of all sources and intermediate results schemas, and \( j \) the set of all intermediate results and the target schemas.

To implement the source-to-target mappings formally defined above, we propose an OWL-based mapping vocabulary: Source-To-Target Mapping (S2TMAP). Figure 6 depicts the mapping vocabulary. A mapping between a source and a target TBox is represented as an instance of the concept map:MapDataset. The source and target TBoxes are defined by instantiating map:TBox, and these
TBoxes are connected to the mapping dataset using the properties `map:sourceTBox` and `map:targetTBox`, respectively. A concept-mapping (an instance of `map:ConceptMapping`) is used to map between a source and a target concepts (instances of `map:Concept`). A concept-mapping is connected to a mapping dataset using the `map:mapDataset` property. The source and target ABox locations of the concept-mapping are defined through the `map:sourceLocation` and `map:targetLocation` properties. The relationship between the concepts can be either `map:subsumption`, `map:supersumption`, `map:Join`, `map:LeftJoin`, `map:RightJoin`, or `map:Equivalence`, and it is connected to the concept-mapping via the `map:relation` property. The sequence of ETL operations, required to implement the concept-mapping at the ABox level, is defined through an RDF sequence. To express joins, the source and target concept in a concept-mapping represent the concepts to be joined, and the join result is stored in the target concept as an intermediate result. In a concept-mapping, we, via `map:commonProperty`, identify the join attributes with a blank node (instance of `map:CommonProperty`) that has, in turn, two properties identifying the source and target join attributes; i.e., `map:commonSourceProperty` and `map:commonTargetProperty`. Since a join can be defined on multiple attributes, we may have multiple blank node definitions. The type of target instance IRIs is stated using the property `map:TargetInstanceIRIType`. If the type is either `map:Property` or `map:Expression`, then the property or expression, to be used to generate the IRIs, is given by `map:targetInstanceIRIvalue`.

To map at the property stage, a property-mapping (an instance of `map:PropertyMapping`) is used. The association between a property-mapping and a concept-mapping is defined by `map:conceptMapping`. The target property of the property-mapping is stated using `map:targetProperty`, and that target property can be mapped with either a source property or an expression. The source type of target property is determined through `map:sourceType4TargetProperty`, and the value is defined by `map:source4TargetPropertyValue`.

**Example 4.** Listing 3 represents a snippet of the mapping file of our use case MD SDW and the source datasets. In the Execution Layer, we show how the different segments of this mapping file will be used by each ETL operation.

A mapping file is a Directed Acyclic Graph (DAG). Figure 7 shows the DAG representation of Listing 3.
In this figure, the sources, intermediate results and the SDW are denoted as nodes of the DAG and edges of the DAG represent the operations. The dotted lines show the parts of the ETL covered by concept-mappings, represented by a rectangle.

6. The Execution Layer

In the Execution Layer, ETL data flows are constructed to populate an MD SDW. Table 1 summarizes the set of ETL operations. In the following, we present an overview of each operation category-wise. Here,
we give the intuitions of the ETL operations in terms of definitions and examples. To reduce the complexity and length of the paper, we place the formal semantics of the ETL operations in the Appendix. In this section, we only present the signature of each operation. That is, the main inputs required to execute the operation. As an ETL data flow is a sequence of operations and an operation in the sequence communicates with its preceding and subsequent operations by means of materialized intermediate results, all the operations presented here have side effects instead of returning output.

Developers can use either of the two following options: (i) The recommended option is that given a TBox construct \( a\text{Construct} \) (a concept, a level, or a QB dataset) and a mapping file \( a\text{Mappings} \) generated in the Definition Layer, the automatic ETL execution flow generation process will automatically extract the parameter values from \( a\text{Mappings} \) (see Section 7 for a detailed explanation of the automatic ETL execution flow generation process). (ii) They can manually set input parameters at the operation level. In this section, we follow the following order to present each operation: 1) we first give a high-level definition of the operation; 2) then, we define how the automatic ETL execution flow generation process parameterizes the operation from the mapping file, and 3) finally, we present an example showing how developers can manually parameterize the operation. When introducing the operations and referring to their automatic parametrization, we will refer to \( a\text{Mappings} \) and \( a\text{Construct} \) as defined here. Note that each operation is bound to exactly one concept-mapping at a time in the mapping file (discussed in Section 7).

### 6.1. Extraction operations

Extraction is process of data retrieval from the sources. Here, we introduce two extraction operations for semantic sources: (i) \( \text{GraphExtractor} \) – to form/extract an RDF graph from a semantic source and (ii) \( \text{TBoxExtraction} \) – to derive a TBox from a semantic source as described in Section 5. As such, \( \text{TBoxExtraction} \) is the only operation in the Execution Layer generating metadata stored in the Mediatory Constructs (see Fig. 5).

\( \text{GraphExtractor}(Q, G, outputPattern, t\text{ABox}) \)

Since the data integration process proposed in this paper uses RDF as the canonical model, we extract/generate RDF triples from the sources with this operation. \( \text{GraphExtractor} \) is functionally equivalent to SPARQL CONSTRUCT queries [39].

If the ETL execution flow is generated automatically, the automatic ETL execution flow generation process first identifies the concept-mapping \( cm \) from \( a\text{Mappings} \) where \( a\text{Construct} \) appears (i.e., in \( a\text{Mapping}, cm \mapsto \text{targetConcept} \mapsto a\text{Construct} \)) and the operation to process \( cm \) is \( \text{GraphExtractor} \) (i.e., \( \text{GraphExtractor} \) is an element in the operation sequence defined by the \text{map:operation} property). Then, it
parametrizes $\text{GraphExtractor}$ as follows: 1) $G$ is the location of the source ABox defined by the property map:sourceLocation of $cm$; 2) $Q$ and outputPattern are internally built based on the map:mappedInstance property, which defines whether all instances (defined by “All”) or a subset of the instances (defined by a filter condition) will be extracted; and 3) $tABox$ is the location of target ABox defined by the property map:targetLocation. A developer can also manually set the parameters. From the given inputs, $\text{GraphExtractor}$ operation performs a pattern matching operation over the given source $G$ (i.e., finds a map function binding the variables in query pattern $Q$ to constants in $G$), and then, for each binding, it creates triples according to the triple templates in outputPattern. Finally, the operation stores the output in the path $tABox$.

**Example 5.** Listing 1 shows the example instances of the Danish Business Dataset (DBD). To extract all instances of bus:Company from the dataset, we use the $\text{GraphExtractor}(Q, G, outputPattern, tABox)$ operation, where

\[
1. Q=((?ins, rdf:type, bus:Company) \land (?ins, ?p, ?v))\]  
2. $G$=“/map/dbd.ttl”,  
3. outputPattern= (?ins, ?p, ?v),  
4. $tABox$=“/map/com.ttl”.

Listing 4 shows the output of this operation.

Listing 4. Example of $\text{GraphExtractor}$

---

6To make it easily distinguishable, here, we use comma instead of space to separate the components of a triple pattern and an RDF triple.

7We present the examples in Turtle format for reducing the space and better understanding. In practice, our system prefers N-Triples format to support scalability.

$\text{TBoxExtraction}$ is already described in Section 5, therefore, we do not repeat it here.

### 6.2 Transformation operations

Transformation operations transform the extracted data according to the semantics of the SDW. Here, we define the following semantic-aware ETL transformation operations: $\text{TransformationOnLiteral}$, $\text{JoinTransformation}$, LevelMemberGenerator, ObservationGenerator, ChangedDataCapture, UpdateLevel, External linking, and MaterializeInference. The following describe each operation.

$\text{TransformationOnLiteral}(sConstruct, tConstruct, sTBox, sABox, propertyMappings, tABox)$ As described in the $\text{SourceToTargetMapping}$ task, a property (in a property-mapping) of a target construct (i.e., a level, a QB dataset, or a concept) can be mapped to either a source concept property or an expression over the source properties. An expression allows arithmetic operations, datatype (string, number, and date) conversion and processing functions, and group functions (sum, avg, max, min, count) as defined in SPARQL [25]. This operation generates the instances of the target construct by resolving the source expressions mapped to its properties.

If the ETL execution flow is generated automatically, the automatic ETL execution flow generation process first identifies the concept-mapping $cm$ from $a\text{Mappings}$, where $a\text{Construct}$ appears and the operation to process $cm$ is $\text{TransformationOnLiteral}$. Then, the process parametrizes $\text{TransformationOnLiteral}$ as follows: 1) $s\text{Construct}$ and $t\text{Construct}$ are defined by $\text{map:sourceConcept}$ and $\text{map:targetConcept}$; 2) $s\text{TBox}$ is the target TBox of $cm$’s map-dataset, defined by the property $\text{map:sourceTBox}$; 3) $s\text{ABox}$ is the location of the source ABox defined by $\text{map:sourceLocation}$; 4) $\text{propertyMappings}$ is the set of property-mappings defined under $cm$; and 5) $t\text{ABox}$ is the location of the target ABox defined by $\text{map:targetLocation}$. A developer can also manually set the parameters. From the given inputs, this operation transforms (or directly returns) the $s\text{ABox}$ triple objects according to the expressions (defined through $\text{map:source4TargetPropertyValue}$ in $\text{propertyMappings}$ and stores the triples in $t\text{ABox}$.

This operation first creates a SPARQL SELECT query based on the expressions defined in $\text{propertyMappings}$, and then, on top of the SELECT query, it forms a SPARQL CONSTRUCT query to generate the transformed ABox for $t\text{Construct}$.
Example 6. Listing 5 (lines 16–19) shows the transformed instances after applying the operation TransformationOnLiteral(sConstruct, tConstruct, sTBox, tABox).

JoinTransformation(sConstruct, tConstruct, sTBox, tBox, sABox, tABox, comProperty, propertyMappings) A TBox construct (a concept, a level, or a property) needs to be enriched with their company information available in the Danish Business dataset. Therefore, a join operation is necessary between sub:Recipient and bus:Company. The concept-mapping of this join is described in Listing 3 at lines 19–37. They are joined by two concept properties: recipient names and their addresses (lines 31, 34–37). We join and transform bus:Company and sub:Recipient using JoinTransformation(sConstruct, tConstruct, sTBox, tBox, sABox, tABox, comProperty, propertyMappings), where

Example 7. The recipients in sdw:Recipient need to be enriched with their company information available in the Danish Business dataset. Therefore, a join operation is necessary between sub:Recipient and bus:Company. The concept-mapping of this join is described in Listing 3 at lines 19–37. They are joined by two concept properties: recipient names and their addresses (lines 31, 34–37). We join and transform bus:Company and sub:Recipient using JoinTransformation(sConstruct, tConstruct, sTBox, tBox, sABox, tABox, comProperty, propertyMappings), where

Listing 5. Example of TransformationOnLiteral

```
1 1. sConstruct = tConstruct = sub:Subsidy,
2 2. sTBox = "map/subsidyTBox.ttl",
3 3. sABox = source instances of sub:Subsidy
4 4. propertyMappings = lines 2–14 in Listing 5,
5 5. tABox = "map/temp1.ttl".
```

Listing 6. Example of JoinTransformation

```
1 1. sConstruct = bus:Company,
2 2. tConstruct = sub:Recipient,
3 3. sTBox = "map/businessTBox.ttl",
4 4. tTBox = "map/subsidyTBox.ttl",
5 5. sABox = source instances of bus:Company
6 6. tABox = source instances of sub:Recipient
7 7. comProperty = lines 31, 34–37 in Listing 3,
8 8. propertyMappings = lines 73–96 in Listing 3.
```
LevelMemberGenerator(sConstruct, level, sTBox, sABox, tTBox, iriValue, iriGraph, propertyMappings, tABox) In QB4OLAP, dimensional data are physically stored in levels. A level member, in an SDW, is described by a unique IRI and its semantically linked properties (i.e., level attributes and roll-up properties). This operation generates data for a dimension schema defined in Definition 1.

If the ETL execution flow is generated automatically, the automatic process first identifies the concept-mapping cm from amappings, where aConstruct appears and the operation to process cm is LevelMemberGenerator. Then it parameterizes LevelMemberGenerator as follows: 1) sConstruct is the source construct defined by map:sourceConcept; 2) level is the target level8 defined by the map:targetConcept property; 3) sTBox and tTBox are the source and target TBoxes of cm’s map dataset, defined by the properties map:sourcesTBox and map:targetTBox; 4) sABox is the source ABox defined by the property map:sourceLocation; 5) iriValue is a rule9 to create IRIs for the level members and it is defined by the map:TargetInstanceIriValue property; 6) iriGraph is the IRI graph10 within which to look up IRIs, given by the developer in the automatic ETL flow generation process; 7) propertyMappings is the set of property-mappings defined under cm; and 8) tABox is the target ABox location defined by map:targetLocation.

A developer can also manually set the parameters. Once it is parameterized, LevelMemberGenerator operation generates QB4OLAP-compliant triples for the level members of level based on the semantics encoded in tTBox and stores them in tABox.

Example 8. Listing 3 shows a concept-mapping (lines 40–54) describing how to populate sdw:Recipient from sub:Recipient. Listing 7 shows the level member created by the LevelMemberGenerator(level, tTBox, sABox, iriValue, iriGraph, propertyMappings, tABox) operation, where

8A level is termed as a level property in QB4OLAP, therefore, throughout this paper, we use both the term “level” and “level property” interchangeably.
9A rule can be either a source property, an expression or incremental, as described in Section 5.
10The IRI graph is an RDF graph that keeps a triple for each resource in the SDW with their corresponding source IRI.

Listing 7. Example of LevelMemberGenerator
3. sTBox = “/map/subsidyTBox.ttl”,
4. sABox = “/map/subsidy.ttl”, shown in Example 7,
5. tTBox = “/map/subsidyMDTBox.ttl”,
6. iriValue = sub:recipientID,
7. iriGraph = “/map/provGraph.nt”,
8. propertyMapping = lines 98–115 in Listing 3,
9. tABox = “/map/temp.ttl”.

ObservationGenerator(sConstruct, dataset, sTBox, sABox, tTBox, iriValue, iriGraph, propertyMappings, tABox) In QB4OLAP, an observation represents a fact. A fact is uniquely identified by an IRI, which is defined by a combination of several members from different levels and contains values for different measure properties. This operation generates data for a cube schema defined in Definition 2.

If the ETL execution flow is generated automatically, the way used by the automatic ETL execution flow generation process to extract values for the parameters of ObservationGenerator from amappings is analogous to LevelMemberGenerator. Developers can also manually set the parameters. Once it is parameterized, the operation generates QB4OLAP-compliant triples for observations of the QB dataset dataset based on the semantics encoded in tTBox and stores them in tABox.

Example 9. Listing 8 (lines 21–25) shows a QB4OLAP-compliant observation create by the ObservationGenerator(sConstruct, dataset, sTBox, sABox, tTBox, iriValue, iriGraph, propertyMappings, tABox) operation, where

1. sConstruct = sub:Subsidy,
2. dataset = sdw:SubsidyMD,
3. sTBox = “/map/subsidyTBox.ttl”
4. sABox = “/map/subsidy.ttl”, shown in Example 6,
5. tTBox = “/map/subsidyMDTBox.ttl”, shown in Listing 2,
6. iriValue = “Incremental”,
therefore, it is out of the scope of this paper. Here, only with the changes of source schemas is an analytical schema and instances. The adaption of the SDW TBox an SDW needs to take action based on the changed both at the schema and instance level. Therefore, world scenario changes occur in a semantic source.

```
ChangedDataCapture(nABox, oABox, flag)
```

This operation must then be connected to either 1) a set of new instances (in the case of SDW dimensional instances in a source), respectively, to (new dimensional instances in a source) and oABox (old dimensional instances in a source), respectively, to parametrize this operation. flag depends on the next operation in the operation sequence.

Developers can also manually set the parameters. From the given inputs, ChangedDataCapture outputs either 1) a set of new instances (in the case of SDW evolution, i.e., flag = 0) or 2) a set of updated triples – the existing triples changed over time – (in the case of SDW update, i.e., flag = 1) and overwrites oABox. This is done by means of the set difference operation. This operation must then be connected to either LevelMemberGenerator to create the new level members or updateLevel (described below) to reflect the changes in the existing level members.

**Example 10.** Suppose Listing 6 is the old ABox of sub:Recipient and the new ABox is at lines 2–10 in Listing 9. This operation outputs either 1) the new instance set (in this case, lines 12–15 in the listing) or 2) the updated triples (in this case, line 17).

```
Listing 9. Example of ChangedDataCapture
```

```
7. iriGraph = "/map/provGraph.nt",
8. propertyMappings = lines 8–19 in Listing 8,
9. tABox = "/map/temp2.ttl"
```

**ChangedDataCapture(nABox, oABox, flag)** In a real-world scenario changes occur in a semantic source both at the schema and instance level. Therefore, an SDW needs to take action based on the changed schema and instances. The adaption of the SDW TBox with the changes of source schemas is an analytical task and requires the involvement of domain experts, therefore, it is out of the scope of this paper. Here, only the changes at the instance level are considered.

If the ETL execution flow is generated automatically, the automatic process first identifies the concept-mapping cm from aMappings, where aConstruct appears and the operation to process cm is ChangedDataCapture. Then, it takes map:sourceLocation and map:targetLocation for nABox (new dimensional instances in a source) and oABox (old dimensional instances in a source), respectively, to parametrize this operation. flag depends on the next operation in the operation sequence.

```
UpdateLevel(level, updatedTriples, sABox, tABox, propertyMappings, iriGraph)
```

Based on the triples updated in the source ABox sABox for the level level (generated by ChangedDataCapture), this operation updates the target ABox tABox to reflect the changes in the SDW according to the semantics encoded in the target TBox tABox and level property-mappings propertyMappings. Here, we address three update types (Type1-update, Type2-update, and Type3-update), defined by Ralph Kimball in [37] for a traditional DW, in an SDW environment. The update types are already defined in tABox for each level attribute of level (as discussed in Section 2.3), so they do not need to be provided as parameters. As we consider only instance level updates, only the objects of the source updated triples are updated. To reflect a source updated triple in level, the level member using the triple to describe itself, will be updated. In short, the level member is updated in the following three ways: 1) A Type1-update simply overwrites the old object with the current object. 2) A Type2-update creates a new version for the level member (i.e., it keeps the previous version and creates a new updated one). It adds the validity interval for both versions. Further, if the level member is a member of an upper level in the hierarchy of a dimension, the changes are propagated downward in the hierarchy, too. 3) A Type3-update
overwrites the old object with the new one. Besides, it adds an additional triple for each changed object to keep the old object. The subject of the additional triple is the instance IRI, the object of the triple is the old object, and the predicate is `concat(oldPredicate, “old-Value”).

If the ETL execution flow is generated automatically, this operation first identifies the concept-mapping `cm` from `aMappings`, where `aConstruct` appears and the operation to process `cm` is `UpdateLevel`. Then it parameterizes `LevelMemberGenerator` as follows: 1) `level` is defined by the `map:targetConcept`; 2) `sABox` is old source data; 3) `updatedTriples` is the source location defined by `map:sourceLocation`; 4) `tBox` is the target `TBox` of `cm`’s map-dataset, defined by the property `map:targetTBox`; 5) `aBox` is defined by `map:targetLocation`; 6) `propertyMappings` is the set of property-mappings defined under `cm`; and 7) `iriGraph` is given by developer in the automatic ETL flow generation process.

Example 11. Listing 10 describes how different types of updates work by considering two members of the `sdw:Recipient` level (lines 2–11). As the name of the second member (lines 7–11) is changed to “Kristian Jensen” from “Kristian Kristensen”, as found in Listing 9. A Type1-update simply overwrites the existing name (line 20). A Type2-update creates a new version (lines 39–46). Both old and new versions contain validity interval (lines 35–37) and (lines 44–46). A Type3-Update overwrites the old name (line 56) and adds a new triple to keep the old name (line 57).

Besides the transformation operations discussed above, we define two additional transformation operations that cannot be run by the automatic ETL dataflows generation process.

**ExternalLinking (sABox, externalSource)** This operation links the resources of `sABox` with the resources of an external source `externalSource`. `externalSource` can either be a SPARQL endpoint or an API. For each resource `inRes` in `sABox`, this operation extracts top k matching external resources either 1) submitting a query to `externalSource` or 2) by sending a web service request embedding `inRes` through the API (e.g., DBpedia lookup API). To find a potential link for each external resource `exRes`, the Jaccard Similarity of the semantic bags of `inRes` and `exRes` is computed. The semantic bag of a resource consists of triples describing the resource [45,46,56]. The pair of the internal and external resources is considered as a match if the Jaccard Similarity exceeds a certain threshold. A triple with the `owl:sameAs` property is created to materialize the link in `sABox` for each pair of matched internal and external resources.

**MaterializeInference(ABox, TBox)** This operation infers new information that has not been explicitly stated in an SDW. It analyzes the semantics encoded into the SDW and enriches the SDW with the inferred triples. A subset of the OWL 2 RL/RDF rules, which encodes part of the RDF-Based Semantics of OWL 2 [52], are considered here. The reasoning rules can

Listing 10. Example of different types of updates
be applied over the TBox \( TBox \) and ABox \( ABox \) separately, and then together. Finally, the resulting inference is asserted in the form of triples, in the same spirit as how the SPARQL regime entailments\(^\text{11}\) deal with inference.

6.3. Load

**Loader(tripleSet, tsPath)** An SDW is represented in the form of RDF triples and the triples are stored in a triplestore (e.g., Jena TDB). Given a set of RDF triples \( \text{triplesSet} \) and the path of a triple store \( \text{tsPath} \), this operation loads \( \text{triplesSet} \) in the triple store.

If the ETL execution flow is generated automatically, this operation first identifies the concept-mapping \( cm \) from \( aMappings \), where \( aConstruct \) appears and the operation to process \( cm \) is \( \text{Loader} \). Then, it takes values of \( \text{map:sourceLocation} \) and \( \text{map:targetLocation} \) for the parameters \( \text{triplesSet} \) and \( \text{tsPath} \).

7. Automatic ETL execution flow generation

We can characterize ETL flows at the Definition Layer by means of the source-to-target mapping file; therefore, the ETL data flows in the Execution Layer can be generated automatically. This way, we can guarantee that the created ETL data flows are sound and relieve the developer from creating the ETL data flows manually. Note that this is possible because the Mediator Constructs (see Fig. 5) contain all the required metadata to automate the process.

Algorithm 1 shows the steps required to create ETL data flows to populate a target construct (a level, a concept, or a dataset). As inputs, the algorithm takes a target construct \( x \), the mapping file \( G \), and the IRI graph \( G_{IRI} \), and it outputs a set of ETL flows \( E \). At first, it uses the functions\(^\text{12}\) to locate \( x \) in \( G \) (line 1) and get the concept-mappings where \( \text{node} \_\text{target} \) participates (line 2). As a (final) target construct can be populated from multiple sources, it can be connected to multiple concept-mappings, and for each concept-mapping, it creates an ETL data flow by calling Algorithm 2: \( \text{CREATEAFLOW} \) (lines 5–8).

---

\(^{11}\)https://www.w3.org/TR/sparql11-entailment/

\(^{12}\)Here, some functions used in the algorithms are characterized by a pattern over \( G \), shown at its side as comments in the corresponding algorithm.

Algorithm 1: \( \text{CREATEETL} \)

**Input:** TargetConstruct \( x \), MappingFile \( G \), IRIGraph \( G_{IRI} \)

**Output:** A set of ETL flows \( E \)

begin

1. \( \text{node}_\text{target} \leftarrow \text{FIND}(x, G) \)
2. \( ?\text{cmaps} \leftarrow \text{FINDCONMAP}(\text{node}_\text{target}, G) \)
   
   / * In \( G \):
   \n   \text{node}_\text{target} \stackrel{\text{map:targetConcept}}{\longrightarrow} ?\text{cmaps} ;
   \n   |?\text{cmaps}| \geq 0 */
3. \( E \leftarrow \text{CREATESET}() \)
4. if \( |?\text{cmaps}| \neq 0 \) then
5.   foreach \( c \in ?\text{cmaps} \) do
6.     \( s_c \leftarrow \text{CREATESTACK}() \)
7.     \( s_c \leftarrow \text{CREATEAFLOW}(c, s_c, G, G_{IRI}) \)
8.     \( E.\text{ADD} (s_c) \)
9. return \( E \)

Algorithm 2: \( \text{CREATEAFLOW} \)

**Input:** ConceptMapping \( c \), Stack \( s_c \), MappingFile \( G \), IRIGraph \( G_{IRI} \)

**Output:** Stack \( s_c \)

begin

1. \( \text{ops} \leftarrow \text{FINDOPERATIONS}(c, G) \)
   
   / * In \( G \):
   \n   \( c \stackrel{\text{map:operation}}{\longrightarrow} \text{ops} */
2. \( s_c.\text{PUSH}() \leftarrow \text{PARAMETERIZE} (\text{ops}, c, G, G_{IRI}) \)
   
   / * Push parameterized operations in \( s_c */
3. \( \text{scon} \leftarrow \text{FINDSOURCECONCEPT}(c, G) \)
   
   / * In \( G \):
   \n   \( c \stackrel{\text{map:sourceConcept}}{\longrightarrow} \text{scon} */
4. \( ?\text{scmap} \leftarrow \text{FINDCONMAP}(\text{scon}, G) \)
   
   / * In \( G \):
   \n   \text{scon} \stackrel{\text{map:targetConcept}}{\longrightarrow} ?\text{scmap} ;
   \n   |?\text{scmap}| \leq 1 */
5. if \( |?\text{scmap}| = 1 \) then
6.   \( \text{CREATEAFLOW}(nc \in ?\text{scmap}, s_c, G) \)
   
   / * recursive call with \( nc */
7. \( s_c.\text{PUSH}() \leftarrow \text{StartOp} \)
   
   / * Push the ETL start operation to \( s_c */
8. return \( s_c \)

Algorithm 2 generates an ETL data flow for a concept-mapping \( c \) and recursively does so if the cur-
Algorithm 3: PARAMETERIZE

Input: Seq \( ops \), ConceptMapping \( cm \),MappingFile \( G \), IRIGraph \( G_{IRI} \)

Output: Stack, \( s_{op} \)

begin
1. \( s_{op} \leftarrow CREATESTACK() \)
2. for \( (i = 1 \text{ to } LENGTH(ops)) \) do
3. if \( (i = 1) \) then
4. PARAMETERIZEOP(\( op[i] \), \( G \), LOC(cm), \( G_{IRI} \))
5. \( s_{op}.PUSH(op[i]) \)
6. PARAMETERIZEOP(\( op[i] \), \( G \), OUTPUTPATH(op[i − 1]), \( G_{IRI} \))
7. \( s_{op}.PUSH(op[i]) \)
8. return \( s_{op} \)

rent concept-mapping source element is connected to another concept-mapping, until it reaches a source element. Algorithm 2 recursively calls itself and uses a stack to preserve the order of the partial ETL data flows created for each concept-mapping. Eventually, the stack contains the whole ETL data flow between the source and target schema.

Algorithm 2 works as follows. The sequence of operations in \( c \) is pushed to the stack after parameterizing it (lines 1–2). Algorithm 3 parameterizes each operation in the sequence, as described in Section 6 and returns a stack of parameterized operations. As inputs, it takes the operation sequence, the concept-mapping, the mapping file, and the IRI graph. For each operation, it uses the PARAMETERIZEOP(\( op \), \( G \), LOC(cm), \( G_{IRI} \)) function to automatically parameterize \( op \) from \( G \) (as all required parameters are available in \( G \)) and push the parameterized operation in a stack (line 2–7).

Note that for the first operation in the sequence, the algorithm uses the source ABox location of the concept-mapping (line 4) as an input, whereas for the remaining operations, it uses the output of the previous operation as input ABox (line 6). Finally, Algorithm 3 returns the stack of parameterized operations.

Then, Algorithm 2 traverses to the adjacent concept-mapping of \( c \) connected via \( c \)’s source concept (line 3–4). After that, the algorithm recursively calls itself for the adjacent concept-mapping (line 6). Note that here, we set a restriction: except for the final target constructs, all the intermediate source and target concepts can be connected to at most one concept-mapping. This constraint is guaranteed when building the meta-data in the Definition Layer. Once there are no more intermediate concept-mappings, the algorithm pushes a dummy starting ETL operation (\( StartOp \)) (line 7) to the stack and returns it. \( StartOp \) can be considered as the root of the ETL data flows that starts the execution process. The stacks generated for each concept-mapping of \( node_{target} \) are added to the output set \( E \) (line 8 in Algorithm 1). The following section shows this process with an example of our use case.

7.1. Auto ETL example

In this section, we show how to populate \( sdw:Recipient \) level using CREATEETL (Algorithm 1). As input, CREATEETL takes the target construct \( sdw:Recipient \), the mapping file Listing 3 and the location of IRI graph “\( /map/provGraph.nt \)”. Figure 8 presents a part (necessary to explain this process) of Listing 3 as an RDF graph. As soon as \( sdw:Recipient \) is found in the graph, the next task is to find the concept-mappings that are connected to \( sdw:Recipient \) through \( map:targetConcept \) (line 2). Here, \( ?cmap = \{ :Recipient_RecipientMD \} \). For \( :Recipient_RecipientMD \), the algorithm creates an empty stack \( s_{c} \) and calls CREATEAFLOW (:\( :Recipient_RecipientMD \), \( s_{c} \), Listing 3) (Algorithm 2) at lines 6–7.

CREATEAFLOW retrieves the sequence of operations (line 1) needed to process the concept-mapping, here: \( ops = \{ LevelMemberGenerator; Loader \} \) (see Fig. 8). Then, CREATEAFLOW parameterizes \( ops \) by calling PARAMETERIZE (Algorithm 3) and then pushes the parameterized operations to \( s_{c} \) (line 2). CREATEETL creates an empty stack \( s_{op} \) (line 1) and for each operation in \( ops \) it calls the PARAMETERIZE() method to parameterize the operation using the concept-mapping information from Listing 3, the source location of the concept-mapping, and the IRI graph, and then it pushes the parameterized operation in \( s_{op} \) (lines 2–7). After the execution of the for loop in PARAMETERIZE (line 2–7), the value of the stack \( s_{op} \) is \( \{ \text{Loader}("/map/temp.nt", "/map/sdw")}; LevelMemberGenerator(sub:Recipient;sdw:Recipient, "/map/subsidyTBox.ttl", "/map/subsidy.nt", "/map/subsidyMDTBox.ttl", sub:recipientID, "map/provGraph.nt", propertyMappings,13 \("/map/temp.nt")\} \), which is returned to line 2 of CREATEAFLOW. Note that the output path of LevelMemberGener
Fig. 8. The graph presentation of a part of Listing 3.

8. Evaluation

We created a GUI-based prototype, named SETLCONSTRUCT [17] based on the different high-level constructs described in Sections 5 and 6. We use Jena 3.4.0 to process, store, and query RDF data and Jena TDB as a triplestore. SETLCONSTRUCT is developed in Java 8. Like other traditional tools (e.g., PDI [11]), SETLCONSTRUCT allows developers to create ETL flows by dragging, dropping, and connecting the ETL operations. The system is demonstrated in [17]. On top of SETLCONSTRUCT, we implement the automatic ETL execution flow generation process discussed in Section 7; we call it SETLAUTO. The source code, examples, and developer manual for SETLCONSTRUCT and SETLAUTO are available at https://github.com/bi-setl/SETL.

To evaluate SETLCONSTRUCT and SETLAUTO, we create an MD SDW by integrating the Danish Business Dataset (DBD) [3] and the Subsidy dataset (https://data.farmsubsidy.org/Old/), described in Section 3. We choose this use case and these datasets for evaluation as there already exists an MD SDW, based on these datasets, that has been programatically created using SETLPROG [44]. Our evaluation focuses on three aspects: 1) productivity, i.e., to what extent SETLCONSTRUCT and SETLAUTO ease the work of a developer when developing an ETL task to process semantic data, 2) development time, the time to develop an ETL process, and 3) performance, the time required to run the process. We run the experiments on a laptop with a 2.10 GHz Intel Core(TM) i7-4600U processor, 8 GB RAM, and Windows 10. On top of that we also present the qualitative evaluation of our approach.

In this evaluation process, we use SETLPROG as our competitive system. We could not directly com-

---

14 ..) indicates that the operation is parameterized.
16 Lines 76–93 in Listing 3.
pare SETLCONSTRUCT and SETLAUTO with traditional ETL tools (e.g., PDI, pygramETL) because they 1) do not support semantic-aware data, 2) are not compatible with the SW technology, and 3) cannot support a data warehouse that is semantically defined. On the other hand, we could also not compare them with existing semantic ETL tools (e.g., PoolParty) because they do not support multidimensional semantics at the TBox and ABox level. Therefore, they do not provide any operations for creating RDF data following multidimensional principles. Nevertheless, SETLPKG supports both semantic and non-semantic source integration, and it uses the relational model as a canonical model. In [44], SETLPKG is compared with PDI to some extent. We also present the summary of that comparison in the following sections. We direct readers to [44] for further details.

8.1. Productivity

SETLPKG requires Python knowledge to maintain and implement an ETL process. On the other hand, using SETLCONSTRUCT and SETLAUTO, a developer can create all the phases of an ETL process by interacting with a GUI. Therefore, they provide a higher level of abstraction to the developer that hides low-level details and requires no programming background. Table 2 summarizes the effort required by the developer to create different ETL tasks using SETLPKG, SETLCONSTRUCT, and SETLAUTO. We measure the developer effort in terms of Number of Typed Characters (NOTC) for SETLPKG and in Number of Used Concepts (NOUC) for SETLCONSTRUCT and SETLAUTO. Here, we define a concept as a GUI component of SETLCONSTRUCT that opens a new window to perform a specific action. A concept is composed of several clicks, selections, and typed characters. For each ETL task, Table 2 lists: 1) its sub construct set, required procedures/data structures, number of the task used in the ETL process (NUEP), and NOTC for SETLPKG; 2) the required task/operation, NOUC, and number of clicks, selections, and NOTC (for each concept) for SETLCONSTRUCT and SETLAUTO. Note that NOTC depends on the user input. Here, we generalize it by using same user input for all systems. Therefore, the total NOTC is not the summation of the values of the respective column.

To create a target TBox using SETLPKG, an ETL developer needs to use the following steps: 1) defining the TBox constructs by instantiating the Concept, Property, and BlankNode classes that play a meta modeling role for the constructs and 2) calling conceptPropertyBinding() and createTriples(). Both procedures take the list of all concepts, properties, and blank nodes as parameters. The former one internally connects the constructs to each other, and the latter creates triples for the TBox. To create the TBox of our SDW using SETLPKG, we used 24,509 NOTC. On the other hand, SETLCONSTRUCT and SETLAUTO use the TargetTBoxDefinition interface to create/edit a target TBox. To create the target TBox of our SDW in SETLCONSTRUCT and SETLAUTO, we use 101 concepts that require only 382 clicks, 342 selections, and 1905 NOTC. Therefore, for creating a target TBox, SETLCONSTRUCT and SETLAUTO use 92% fewer NOTC than SETLPKG.

To create source-to-target mappings, SETLPKG uses Python dictionaries where the keys of each dictionary represent source constructs and values are the mapped target TBox constructs, and for creating the ETL process it uses 23 dictionaries, where the biggest dictionary used in the ETL process takes 253 NOTC. In total, SETLPKG uses 2052 NOTC for creating mappings for the whole ETL process. On the contrary, SETLCONSTRUCT and SETLAUTO use a GUI. In total, SETLCONSTRUCT uses 87 concepts composed of 330 clicks, 358 selections, and 30 NOTC. Since SETLAUTO internally creates the intermediate mappings, there is no need to create separate mapping datasets and concept-mappings for intermediate results. Thus, SETLAUTO uses only 65 concepts requiring 253 clicks, 274 selections, and 473 NOTC. Therefore, SETLCONSTRUCT reduces NOTC of SETLPKG by 98%. Although SETLAUTO uses 22 less concepts than SETLCONSTRUCT, SETLCONSTRUCT reduces NOTC of SETLAUTO by 93%. This is because, in SETLAUTO, we write the data extraction queries in the concept-mappings where in SETLCONSTRUCT we set the data extraction queries in the ETL operation level.

To extract data from either an RDF local file or an SPARQL endpoint, SETLPKG uses the query() procedure and the ExtractTriplesFromEndpoint() class. On the other hand, SETLCONSTRUCT uses the GraphExtractor operation. It uses 1 concept composed of 5 clicks and 20 NOTC for the local file and 1 concept with 5 clicks and 30 NOTC for the endpoint. SETLPKG uses different functions/procedures from the Petl Python library (integrated with SETLPKG) based on the cleansing requirements. In SETLCONSTRUCT, all data cleansing related tasks on data sources are done using TransformationOnLiteral (single source) and JoinTransformation (for multi-source). TransformationOnLiteral
<table>
<thead>
<tr>
<th>ETL Task</th>
<th>Sub construct</th>
<th>Procedures/Data structures</th>
<th>NOUC</th>
<th>Task/Operation</th>
<th>NOTC</th>
<th>Components of Each Concept</th>
<th>NOUC</th>
<th>Components of Each Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBox with MD Semantics</td>
<td>Cube Structure</td>
<td>Concept(), Property(), BlankNode(), Namespace(), conceptPropertyBindings(), createOntology()</td>
<td>1</td>
<td>TargetTBox-Definition</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Cube Dataset</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>18</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Dimension</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Hierarchy</td>
<td></td>
<td></td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>18</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Level</td>
<td></td>
<td></td>
<td>16</td>
<td>4</td>
<td>5</td>
<td>17</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>Level Attribute</td>
<td></td>
<td></td>
<td>38</td>
<td>4</td>
<td>3</td>
<td>15</td>
<td>38</td>
<td>4</td>
</tr>
<tr>
<td>Rollup Property</td>
<td></td>
<td></td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Measure Property</td>
<td></td>
<td></td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>25</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Hierarchy Step</td>
<td></td>
<td></td>
<td>12</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>Prefix</td>
<td></td>
<td></td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>48</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td><strong>Subtotal (for the Target TBox)</strong></td>
<td></td>
<td></td>
<td>24509</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping Generation</td>
<td>Mapping Dataset</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
<td>SourceTo-TargetMapping</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Concept-mapping</td>
<td>dict()</td>
<td></td>
<td>17</td>
<td>243</td>
<td></td>
<td>23</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Property-mapping</td>
<td>N/A</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>59</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Subtotal (for the Mapping File)</strong></td>
<td></td>
<td></td>
<td>2052</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semantic Data Extraction from an RDF Dump File</td>
<td>–</td>
<td>query()</td>
<td>17</td>
<td>40</td>
<td>GraphExtractor</td>
<td>17</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Semantic Data Extraction through a SPARQL Endpoint</td>
<td>–</td>
<td>ExtractTriplesFrom-Endpoint()</td>
<td>0</td>
<td>100</td>
<td>GraphExtractor</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Cleansing</td>
<td>–</td>
<td>Built-in Petl functions</td>
<td>15</td>
<td>80</td>
<td>Transformation-OnLiteral</td>
<td>5</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Join</td>
<td>–</td>
<td>Built-in Petl functions</td>
<td>1</td>
<td>112</td>
<td>Join-Transformation</td>
<td>1</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>ETL Task</td>
<td>Sub construct</td>
<td>Procedures/Data structures</td>
<td>NUEP</td>
<td>NOTC</td>
<td>Task/Operation</td>
<td>NOUC Components of Each Concept</td>
<td>NOTC Components of Each Concept</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------</td>
<td>------</td>
<td>------</td>
<td>-----------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td></td>
</tr>
<tr>
<td>Level Member Generation</td>
<td>–</td>
<td>createDataTripleToFile(), createDataTripleToTripleStore()</td>
<td>16</td>
<td>75</td>
<td>LevelMember-Generator</td>
<td>16</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Observation Generation</td>
<td>–</td>
<td>createDataTripleToFile(), createDataTripleToTripleStore()</td>
<td>1</td>
<td>75</td>
<td>Observation-Generator</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Loading as RDF Dump</td>
<td>–</td>
<td>insertTriplesIntoTDB(), bulkLoadToTDB()</td>
<td>0</td>
<td>113</td>
<td>Loader</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Loading to a Triplestore</td>
<td>–</td>
<td>insertTriplesIntoTDB(), bulkLoadToTDB()</td>
<td>17</td>
<td>153</td>
<td>Loader</td>
<td>17</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Auto ETL Generation</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>CreateETL</td>
<td>1</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td><strong>Total (for the Execution Layer Tasks)</strong></td>
<td></td>
<td></td>
<td><strong>1</strong></td>
<td><strong>2807</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Grand Total (for the Whole ETL Process)</strong></td>
<td></td>
<td></td>
<td><strong>1</strong></td>
<td><strong>29358</strong></td>
<td></td>
<td><strong>57</strong></td>
<td><strong>991</strong></td>
<td><strong>142</strong></td>
</tr>
</tbody>
</table>
Table 3

Comparison between the ETL processes of SETLPROG and PDI for SDW (Reproduced from [44])

<table>
<thead>
<tr>
<th>Task</th>
<th>Tools</th>
<th>SETL</th>
<th>LOC</th>
<th>PDI (Kettle)</th>
<th>Used tools</th>
<th>Used languages</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBox with MD semantics</td>
<td>Built-in SETL</td>
<td>Python</td>
<td>312</td>
<td>Protege, SETL</td>
<td>Python</td>
<td></td>
<td>312</td>
</tr>
<tr>
<td>Ontology Parser</td>
<td>Built-in SETL</td>
<td>Python</td>
<td>2</td>
<td>User Defined Class</td>
<td>Java</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>Semantic Data Extraction through SPARQL endpoint</td>
<td>Built-in SETL</td>
<td>Python, SPARQL</td>
<td>2</td>
<td>Manually extraction using SPARQL endpoint</td>
<td>SPARQL</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Semantic Data Extraction from RDF Dump file</td>
<td>Built-in SETL</td>
<td>Python</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Reading CSV/Database</td>
<td>Built-in Petl</td>
<td>Python</td>
<td>2</td>
<td>Drag &amp; Drop</td>
<td>NA</td>
<td>Number of used Activities: 1</td>
<td></td>
</tr>
<tr>
<td>Cleansing</td>
<td>Built-in Petl</td>
<td>Python</td>
<td>36</td>
<td>Drag &amp; Drop</td>
<td>NA</td>
<td>Number of used Activities: 19</td>
<td></td>
</tr>
<tr>
<td>IRI Generation</td>
<td>Built-in SETL</td>
<td>Python</td>
<td>2</td>
<td>User Defined Class</td>
<td>Java</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Triple Generation</td>
<td>Built-in SETL</td>
<td>Python</td>
<td>2</td>
<td>User Defined Class</td>
<td>Java</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>External Linking</td>
<td>Built-in SETL</td>
<td>Python</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Loading as RDF dump</td>
<td>Built-in SETL</td>
<td>Python</td>
<td>1</td>
<td>Drag &amp; Drop</td>
<td>NA</td>
<td>Number of used Activities: 1</td>
<td></td>
</tr>
<tr>
<td>Loading to Triple Store</td>
<td>Built-in SETL</td>
<td>Python</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

Total LOC for the complete ETL process: 401

471 LOC + 4 NA + 21 Activities

8.1.1. Comparison between SETLPROG and PDI

PDI is a non-semantic data integration tool that contains a rich set of data integration functionality to create an ETL solution. It does not support any functionality for semantic integration. In [44], we used PDI in combination with other tools and manual tasks to create a version of an SDW. The comparison between the ETL processes of SETLPROG and PDI to create this SDW are shown in Table 3. Here, we outline the developer efforts in terms of used tools, languages, and Lines of Codes (LOC). As mentioned earlier, SETLPROG provides built-in classes to annotate MD constructs with a TBox. PDI does not support defining a TBox. To create the SDW using PDI, we use the TBox created by SETLPROG. SETLPROG provides built-in classes to parse a given TBox and users can use different methods to parse the TBox based on their requirements. In PDI, we implement a Java class to parse the TBox created by SETLPROG, which takes an RDF file containing the definition of a TBox as input and outputs the list of concepts and properties contained in the TBox. PDI is a non-semantic data integration tool, thus, it does not support processing semantic data. We manually extract data from SPARQL endpoints and materialize them in a relational database for further processing. PDI provides activities (drag & drop functionality) to pre-process database and CSV files. On the other hand, SETLPROG provides methods...
to extract semantic data either from a SPARQL endpoint or an RDF dump file batch-wise. In SETLPROG, users can create an IRI by simply passing arguments to the createIRI() method. PDI does not include any functionality to create IRIs for resources. We define a Java class of 23 lines to enable the creation of IRIs for resources. SETLPROG provides the createTriple() method to generate triples from the source data based on the MD semantics of the target TBox; users can just call it by passing required arguments. In PDI, we develop a Java class of 60 lines to create the triples for the sources. PDI does not support to load data directly to a triple store which can easily be done by SETLPROG. Finally, we could not run the ETL process of PDI automatically (i.e., in a single pass) to create the version of a SDW. We instead made it with a significant number of user interactions. In total, SETLPROG takes 401 Lines of Code (LOC) to run the ETL, where PDI takes 471 LOC + 4 Not Applicable (N/A) + 21 Activities. Thus, SETLPROG creates the SDW with 14.9% less LOC and minimum user interactions comparing to PDI where users have to build their own Java classes, plug-in, and manual tasks to enable semantic integration. Therefore, we can conclude that SETLPROG uses 14.9% less LOC than PDI, SETLCONSTRUCT uses 92% fewer NOTC than SETLPROG, and SETLAUTO further reduces NOUC by another 25%. Combining these results, we see that SETLCONSTRUCT and SETLAUTO have significantly better productivity than PDI for building an SDW.

8.2. Development time

We compare the time used by SETLPROG, SETLCONSTRUCT, and SETLAUTO to build the ETL processes for our use case SDW. As the three tools were developed within the same project scope and we master them, the first author conducted this test. We chose the running use case used in this paper and created a solution in each of the three tools and measured the development time. We used each tool twice to simulate the improvement we may obtain when we are knowledgeable about a given project. The first time it takes more time to analyze, think, and create the ETL process, and in the latter, we reduce the interaction time spent on analysis, typing, and clicking. Table 4 shows the development time (in minutes) for main integration tasks used by the different systems to create the ETL processes.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Iteration number</th>
<th>Target TBox definition</th>
<th>Mapping generation</th>
<th>ETL design</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETLPROG</td>
<td>1</td>
<td>186</td>
<td>51</td>
<td>85</td>
<td>322</td>
</tr>
<tr>
<td>SETLPROG</td>
<td>2</td>
<td>146</td>
<td>30</td>
<td>65</td>
<td>241</td>
</tr>
<tr>
<td>SETLCONSTRUCT</td>
<td>1</td>
<td>97</td>
<td>46</td>
<td>35</td>
<td>178</td>
</tr>
<tr>
<td>SETLCONSTRUCT</td>
<td>2</td>
<td>58</td>
<td>36</td>
<td>30</td>
<td>124</td>
</tr>
<tr>
<td>SETLAUTO</td>
<td>1</td>
<td>97</td>
<td>40</td>
<td>2</td>
<td>139</td>
</tr>
<tr>
<td>SETLAUTO</td>
<td>2</td>
<td>58</td>
<td>34</td>
<td>2</td>
<td>94</td>
</tr>
</tbody>
</table>

SETLPROG took twice as long as SETLCONSTRUCT and SETLAUTO to develop a target TBox. In SETLPROG, to create a target TBox construct, for example a level l, we need to instantiate the Concept() concept for l and then add its different property values by calling different methods of Concept(). SETLCONSTRUCT and SETLAUTO create l by typing the input or selecting from the suggested items. Thus, SETLCONSTRUCT and SETLAUTO also reduce the risk of making mistakes in an error-prone task, such as creating an ETL. In SETLPROG, we typed all the source and target properties to create a mapping dictionary for each source and target concept pair. However, to create mappings, SETLCONSTRUCT and SETLAUTO select different constructs from a source as well as a target TBox and only need to type when there are expressions and/or filter conditions of queries. Moreover, SETLAUTO took less time than SETLCONSTRUCT in mapping generation because we did not need to create mappings for intermediate results.

To create ETL data flows using SETLPROG, we had to write scripts for cleansing, extracting, transforming, and loading. SETLCONSTRUCT creates ETL flows using drag-and-drop options. Note that the mappings in SETLCONSTRUCT and SETLAUTO use expressions to deal with cleansing and transforming related tasks; however, in SETLPROG we cleansed and transformed the data in the ETL design phase. Hence, SETLPROG took more time in designing ETL compared to SETLCONSTRUCT. On the other hand, SETLAUTO creates the ETL data flows automatically from a given mapping file and the target TBox. Therefore, SETLAUTO took only two minutes to create the flows. In short, SETLPROG is a programmatic environment, while SETLCONSTRUCT and SETLAUTO are drag and drop tools. We exemplify this fact by means of Figs 9 and 10, which showcase the creation of the ETL.
Fig. 9. The segment of the main method to populate the `sdw:SubsidyMD` dataset and the `sdw:Recipient` using SETLPROG.

```python
if __name__ == "_main_":
    datawarehouse.createOntology()  # TBox creation
    query(localRDFfile='C:/Experiment/recipient.nt', query='select * where { ?a ?b ?c}.
        outpaths='C:/Experiment/temp/Recipient.csv')  # Extraction from recipient.nt
    query(localRDFfile='C:/Experiment/company.nt', query='select * where { ?a ?b ?c}.
        outpaths='C:/Experiment/temp/Company.csv')  # Extraction from company.nt
    query(localRDFfile='C:/Experiment/subsidy.nt', query='select * where { ?a ?b ?c}.
        outpaths='C:/Experiment/temp/subsidy.csv')  # Extraction from subsidy.nt
    recipient=cleaner.fromcsv('C:/Experiment/temp/Recipient.csv')  # reading from Recipient.csv
    company=cleaner.fromcsv('C:/Experiment/temp/Company.csv')  # reading from Recipient.csv
    Recipient=cleaner.outJoin(recipient, company,
        keys=['name', 'address'])  # join recipient and company
    recipient=cleaner.cut(Recipient, 'recipientid', 'recipientname', 'hastown', 'address',
        'companyid', 'hasFormat', 'mainActivity', 'secondaryActivity', 'globalType')
    cleaner.tocsv(Recipient, 'C:/Experiment/temp/recipient_final.csv')  # writing to a CSV file
    mapping_recipient=itertools()  # creating a Python dictionary for recipient
    mapping_subsidy=itertools()  # creating a Python dictionary for subsidy
    mapping_recipient=('hastown': mdProperty:inCity, 'recipientid': mdAttribute:recipientid,
        'recipientname': mdAttribute:recipientname, 'address': mdAttribute:address,
        'companyid': mdAttribute:companyid, 'globalType': mdAttribute:globalType)
    mapping_subsidy=('amount': mdProperty:amount, 'year': mdProperty:year,
        'hasRecipient': mdProperty:hasRecipient, 'source': mdProperty:source,
        'hasSubsidy': mdProperty:hasSubsidy, 'budgetline': mdProperty:budgetline)
    start=giveSecond(time.localtime())
    file=csv.DictReader(open('C:/Experiment/temp/recipient_final.csv',
        encodings='utf8'))  # creating a dictionary from recipient_final.csv
    name='Recipient'
    x.createTripleToFile(file, sdw, 'recipientid', mapping_recipient, classlist,
        propertylist, 'a')  # create RDF triples for Recipient Level
    file=csv.DictReader(open('C:/Experiment/temp/subsidy.csv',
        encodings='utf8'))
    name='Subsidy'
    x.createTripleToFile(file, sdw, 'subsidyid', mapping_subsidy, classlist,
        propertylist, 'a')  # create RDF triples for Subsidy dataset
    start2=giveSecond(time.localtime())
    print("Time to create RDF data = " + str(start2-start))
```

Fig. 10. The ETL flows to populate the `sdw:SubsidyMD` dataset and the `sdw:Recipient` level using SETLCONSTRUCT.

In summary, using `SETLCONSTRUCT`, the development time is cut in almost half (41% less development time than `SETLPROG`); and using `SETLAUTO`, it is cut by another 27%.
Table 5

<table>
<thead>
<tr>
<th>Performance metrics</th>
<th>Systems</th>
<th>Extraction and traditional transformation</th>
<th>Semantic transformation</th>
<th>Loading</th>
<th>Total processing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(in minutes)</td>
<td>SETL-PROG</td>
<td>33</td>
<td>17.86</td>
<td>21</td>
<td>71.86</td>
</tr>
<tr>
<td></td>
<td>SETL-CONSTRUCT</td>
<td>43.05</td>
<td>39.42</td>
<td>19</td>
<td>101.47</td>
</tr>
<tr>
<td>Input size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SETL-PROG</td>
<td>6.2 GB (Jena TDB) + 6.1 GB (N-Triples)</td>
<td>496 MB (CSV)</td>
<td>4.1 GB (N-Triples)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SETL-CONSTRUCT</td>
<td>6.2 GB (Jena TDB) + 6.1 GB (N-Triples)</td>
<td>6.270 GB (N-Triples)</td>
<td>4.1 GB (N-Triples)</td>
<td></td>
</tr>
<tr>
<td>Output size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SETL-PROG</td>
<td>490 MB (CSV)</td>
<td>4.1 GB (N-Triples)</td>
<td>3.7 GB (Jena TDB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SETL-CONSTRUCT</td>
<td>6.270 GB (N-Triples)</td>
<td>4.1 GB (N-Triples)</td>
<td>3.7 GB (Jena TDB)</td>
<td></td>
</tr>
</tbody>
</table>

8.3. Performance

Since the ETL processes of SETL-CONSTRUCT and SETL-AUTO are the same and only differ in the developer effort needed to create them, this section only compares the performance of SETL-PROG and SETL-CONSTRUCT. We do so by analyzing the time required to create the use case SDW by executing the respective ETL processes. To evaluate the performance of similar types of operations, we divide an ETL process into three sub-phases: extraction and traditional transformation, semantic transformation, as well as loading and discuss the time to complete each.

Table 5 shows the processing time (in minutes), input and output size of each sub-phase of the ETL processes created by SETL-PROG and SETL-CONSTRUCT. The input and output formats of each sub-phase are shown in parentheses. The extraction and traditional transformation sub-phases in both systems took more time than the other sub-phases. This is because they include time for 1) extracting data from large RDF files, 2) cleansing and filtering the noisy data from the DBD and Subsidy datasets, and 3) joining the DBD and Subsidy datasets. SETL-CONSTRUCT took more time than SETL-PROG because its TransformationOnLiteral and JoinTransformation operations use SPARQL queries to process the input file whereas SETL-PROG uses the methods from the Petl Python library to cleanse the data extracted from the sources.

SETL-CONSTRUCT took more time during the semantic transformation than SETL-PROG because SETL-CONSTRUCT introduces two improvements over SETL-PROG: 1) To guarantee the uniqueness of an IRI, before creating an IRI for a target TBox construct (e.g., a level member, an instance, an observation, or the value of an object or roll-up property), the operations of SETL-CONSTRUCT search the IRI provenance graph to check the availability of an existing IRI for that TBox construct. 2) As input, the operations of SETL-CONSTRUCT take RDF (N-triples) files that are larger in size than the CSV files (see Table 5), used by SETL-PROG as input format. To ensure our claims, we run an experiment for measuring the performance of the semantic transformation procedures of SETL-PROG and the operations of SETL-CONSTRUCT by excluding the additional two features introduced in SETL-CONSTRUCT operations (i.e., a SETL-CONSTRUCT operation does not lookup the IRI provenance graph before creating IRIs and takes a CSV input). Figure 11 shows the processing time taken by SETL-CONSTRUCT operations and SETL-PROG procedures to create level members and observations with increasing input size. In the figure, LG and OG represent level member generator and observation generator operations (in case of SETL-CONSTRUCT) or procedures (in case of SETL-PROG).

In summary, to process an input CSV file with 500 MB in size, SETL-CONSTRUCT takes 37.4% less time than SETL-PROG to create observations and 23.4% less time than SETL-PROG to create level members. The figure also shows that the processing time difference between the corresponding SETL-CONSTRUCT op-
eration and the \textit{SETLPROG} procedure increases with the size of the input. In order to guarantee scalability when using the Jena library, a \textit{SETLCONSTRUCT} operation takes the (large) RDF input in the N-triple format, divides the file into several smaller chunks, and processes each chunk separately. Figure 12 shows the processing time taken by \textit{LevelMemberGenerator} and \textit{ObservationGenerator} operations with the increasing number of triples. We show the scalability of the \textit{LevelMemberGenerator} and \textit{ObservationGenerator} because they create data with MD semantics. The figure shows that the processing time of both operations increase linearly with the increase in the number of triples, which ensures that both operations are scalable.

\textit{SETLCONSTRUCT} takes less time in loading than \textit{SETLPROG} because \textit{SETLPROG} uses the Jena TDB loader command to load the data while \textit{SETLCONSTRUCT} programmatically load the data using the Jena API's method.

In summary, \textit{SETLPROG} and \textit{SETLCONSTRUCT} have similar performance (29% difference in total processing time). \textit{SETLCONSTRUCT} ensures the uniqueness of IRI creation and uses RDF as a canonical model, which makes it more general and powerful than \textit{SETLPROG}.

Besides the differences of the performances already explained in Table 5, \textit{SETLCONSTRUCT} also includes an operation to update the members of the SDW levels, which is not included in \textit{SETLPROG}. Since the ETL process for our use case SDW did not include that operation, we scrutinize the performance of this specific operation of \textit{SETLCONSTRUCT} in the following.

\textbf{Performance analysis of UpdateLevel operation} Figure 13 shows the performance of the \textit{UpdateLevel} operation. To evaluate this operation, we consider two levels: \textit{mdProperty:Recipient} and \textit{mdProperty:City}. We consider these two levels because \textit{mdProperty:Recipient} is the largest level (in terms of size) in our use case, and \textit{mdProperty:City} is the immediate upper level of \textit{mdProperty:Recipient} in the \textit{mdStructure:Address} hierarchy of the dimension \textit{mdProperty:Beneficiary}. Therefore, we can record how the changes in cities propagate to recipients, especially in Type2-update. The \textit{sdw:Recipient} level is the lowest granularity level in the \textit{mdStructure:Address} hierarchy; therefore, changes in a recipient (i.e., a member of \textit{sdw:Recipient}) only affect that recipient. Figure 13a shows the processing time with the increasing number of recipients. As a Type2-update creates a new version for each changed level member, it takes more time than a Type1-update and a Type3-update. A Type3-update takes more time than a Type1-update because it keeps the record of old property values besides the current ones. Figure 13b shows how the size of the target ABox increases with the increasing number of recipients. The target ABox size increases linearly with the increasing number of recipients (see Fig. 13b) for Type2-update and Type3 updates because they keep additional information. However, the target ABox size decreases with the increasing number of recipients for Type1-updates; this is because the current property-values are smaller than the older ones in size.

Figure 13c shows the processing time when increasing the number of updated cities. Since \textit{sdw:City} is the immediate upper level of \textit{sdw:Recipient} in the \textit{mdStructure:address} hierarchy, to reflect a changed city in the target ABox, a Type2-update creates new versions for itself as well as for the recipients living in that city. Therefore, a Type2-update takes more processing time in comparison to a Type1-update and a Type-3 update. Figure 13d shows the target ABox size with the increasing number of cities. The figure shows that the target ABox size for Type2-updates increases slowly within the range from 120 to 160 (X-axis), and it indicates that the changed cities within this range contain fewer recipients than the cities in other ranges.

\subsection{Comparison between SETLPROG and PDI in processing \textit{sdw:Subsidy}}

In [44], we compare the performance of \textit{SETLPROG} with a non-semantic data integration tool, PDI. We populate the SDW for \textit{sdw:Subsidy} concept using PDI. To run the process PDI takes 1903 s. On other hand, SETL takes only 1644 s. Thus, SETL is 259 s (13.5%) faster than PDI. PDI is much slower because
almost everything MD-related must be embedded as UDFs or equivalent elements.

In overall, we conclude that SETLPROG is better in performance compared to PDI and SETLCONSTRUCT. However, SETLCONSTRUCT is a more correct and scalable framework. On top of that, SETLCONSTRUCT supports SDWs update.

8.4. Qualitative evaluation

To evaluate our proposed high-level ETL qualitatively, we selected and interviewed two experts with expertise across traditional ETL and semantic integration. We find that two experts are sufficient, because both of them have 10+ years research experiences in ETL and are the architects of popular open-source ETL tools (pygramETL [60] and QUARRY [33]). In this section, we present a high-level summary of the expert interviews. We follow the procedure described in [10, 42] to structure the questionnaire and the answers of the experts. The two experts participated in the interviews via email. Interviews included nine semi-structured questions\(^{17}\) aimed at gathering expert opinions on the significance of the proposed two-layered RDF-based semantic data integration paradigm, completeness and soundness of the proposed set of ETL operations, and further possibilities of the proposed approach. Table 6 shows the questions, their categories, and the responses as well as justifications of both experts. A response is rated from 1 to 5 (1 – strongly disagree, 2 – disagree, 3 – neutral, 4 – agree, and 5 - strongly agree). Q4, Q7, Q8, and Q9 contain free-text answers only.

On the significance of the proposed paradigm category, both experts provide positive responses to all questions. Their only suggestion is to allow technical users to extend the metadata artifacts because ETL tool palettes can typically be much more complex. We use RDF as the canonical model in our paradigm, and it allows users to extend metadata artifacts based on their business requirements. Both experts think that our pro-

\(^{17}\)https://drive.google.com/file/d/1qgtjinXtOah5SR8LyoF3e6oQzGZdZ/view?usp=sharing
Table 6
Summary of the expert interviews

<table>
<thead>
<tr>
<th>Category</th>
<th>No.</th>
<th>Questions</th>
<th>Expert 1</th>
<th>Expert 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Response</td>
<td>Response</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Justification</td>
<td>Justification</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Significance of the proposed paradigm</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Q1</td>
<td>Is the two-layered paradigm followed by SETLCONSTRUCT more convenient</td>
<td>Separation of</td>
<td>The intervention of more</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by SETLCONSTRUCT more convenient than the traditional approach?</td>
<td>Concerns</td>
<td>technical users is required</td>
</tr>
<tr>
<td></td>
<td>Q2</td>
<td>Are the metadata artifacts from Definition Layer sufficient to deal with</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real-case problems?</td>
<td>Extensibility</td>
<td>Complete for the ETL specific</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>by the user</td>
<td>to semantic data and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>would be</td>
<td>multidimensional analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>useful</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q3</td>
<td>Do you consider SETLCONSTRUCT’s target-driven approach to build semantic</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data warehouses appropriate?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q4</td>
<td>Do you consider scon/is a valuable and notable contribution compared to</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>traditional ETL tools and semantic-oriented ETL tools?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q5</td>
<td>Is the set of ETL operations proposed sound?</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Okay</td>
<td>The ETL operations seem to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>be correct</td>
</tr>
<tr>
<td></td>
<td>Q6</td>
<td>Are the proposed operations complete?</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sufficient</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q7</td>
<td>Do you miss anything (else) in scon current set of core ETL operations</td>
<td>UDFs would be useful</td>
<td>Row normalizer/denormalizer</td>
</tr>
<tr>
<td></td>
<td>Q8</td>
<td>Do you consider automation of the mapping file is feasible?</td>
<td>Semi-automation is feasible</td>
<td>Not easy to express more complex, black box operations</td>
</tr>
<tr>
<td>Further possibilities</td>
<td>Q9</td>
<td>Do you think our approach fits in the context of Big Data and Data Lakes</td>
<td>Yes</td>
<td>Limited to support complex data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(and how)?</td>
<td></td>
<td>transformations in the Big Data context</td>
</tr>
</tbody>
</table>

Proposed ETL operations are sound and complete considering that the ETL is specific to semantic data and multidimensional analysis. However, one of the suggestions is to make the set of ETL operations open and to introduce user-defined functions (UDFs) so that users can write their own functions if there is a requirement for additional transformations (e.g., row normalizer/denormalizer, language conversion). In our future work, we will introduce UDFs in the set of ETL operations. Both experts do not support complete automation of the mapping file considering the complex nature of ETL scenarios. However, they think that semi-automation is feasible. Expert 1 thinks that we can apply this two-layered RDF-based data integration approach in the context of Big data and Data Lakes. However, Expert 2 thinks that there is a need to support more complex data transformations.

In summary, both experts acknowledge the contribution of our work and think that the set of ETL operations is complete and sound to annotate a knowledge base with multidimensional semantics. However, to fit this approach in the context of Big Data and data lakes, there is a need to include UDFs to support more complex transformations.

9. Related work

Nowadays, combining SW and BI technologies is an emerging research topic as it opens interesting research opportunities. As a DW deals with both internal and (increasingly) external data presented in heterogeneous formats, especially in the RDF format, semantic issues should be considered in the integration process [1]. Furthermore, the popularity of SW data gives rise to new requirements for BI tools to enable OLAP-style analysis over this type of data [32]. Therefore, the existing research related to semantic ETL is divided into two lines: 1) on the one hand, the use of SW technologies to physically integrate heterogeneous sources and 2) on the other hand, enabling OLAP analysis over SW data.
One prominent example following the first research line is [58], which presents an ontology-based approach to enable the construction of ETL flows. At first, the schema of both the sources and the DW are defined by a common graph-based model, named the datastore graph. Then, the semantics of the datastore graphs of the data sources and the DW are described by generating an (OWL-based) application ontology, and the mappings between the sources and the target are established through that ontology. In this way, this approach addresses heterogeneity issues among the source and target schemata and finally demonstrates how the use of an ontology enables a high degree of automation from the source to the target attributes, along with the appropriate ETL transformations. Nonetheless, computationally complex ETL operations like slowly changing dimensions and the annotation of the application ontology with MD semantics are not addressed in this work. Therefore, OLAP queries cannot be applied on the generated DW.

Another piece of existing work [6] aligned to this line of research proposes a methodology describing some important steps required to make an SDW, which enables to integrate data from semantic databases. This approach also misses the annotation of the SDW with MD semantics. [59] has proposed an approach to support data integration tasks in two steps: 1) constructing ontologies from XML and relational sources and 2) integrating the derived ontologies by means of existing ontology alignment and merging techniques. However, ontology alignment techniques are complex and error-prone. [5] presents a semantic ETL framework at the conceptual level. This approach utilizes the SW technologies to facilitate the integration process and discusses the use of different available tools to perform different steps of the ETL process. [3] presents a method to spatially integrate a Danish Agricultural dataset and a Danish Business dataset using an ontology. The approach uses SQL views and other manual processes to cleanse the data and Virtuoso for creating and storing integrated RDF data. [38] presents UnifiedViews, an open-source ETL framework that supports management of RDF data. Based on the SPARQL queries, they define four types of Data Processing Units (DPUs): Extractor, Transformer, Loader, and Quality Assessor. However, the DPUs do not support to generate MD RDF data.

In the second line of research, a prominent paper is [47], which outlines a semi-automatic method for incorporating SW data into a traditional MD data management system for OLAP analysis. The proposed method allows an analyst to accomplish the following tasks: 1) designing the MD schema from the TBox of an RDF dataset, 2) extracting the facts from the ABox of the dataset and populating the MD fact table, and 3) producing the dimension hierarchies from instances of the fact table and the TBox to enable MDX queries over the generated DW. However, the generated DW no longer preserves the SW data principles defined in [27]; thus, OLAP analysis directly over SW data is yet to be addressed. To address this issue, [13] introduces the notion of a lens, called the analytical schema, over an RDF dataset. An analytical schema is a graph of classes and properties, where each node of the schema presents a set of facts that can be analyzed by traversing the reachable nodes. [28] presents a self-service OLAP endpoint for an RDF dataset. This approach first superimposes an MD schema over the RDF dataset. Then, a semantic analysis graph is generated on top of that MD schema, where each node of the graph represents an analysis situation corresponding to an MD query, and an edge indicates a set of OLAP operations.

Both [13] and [28] require either a lens or a semantic analysis graph to define MD views over an RDF dataset. Since most published SW data contains facts and figures, W3C recommends the Data Cube (QB) [15] vocabulary to standardize the publication of SW data with MD semantics. Although QB is appropriate to publish statistical data and several publishers (e.g., [51]) have already used the vocabulary for publishing statistical datasets, it has limitations to define MD semantics properly. The QB4OLAP [20] vocabulary enriches QB to support MD semantics by providing constructs to define 1) a cube structure in terms of different level of dimensions, measures, and attaching aggregate functions with measures and 2) a dimension structure in terms of levels, level attributes, relationships, the cardinality of relationships among the levels, and hierarchies of the dimension. Therefore, MD data can be published either by enriching data already published using QB with dimension levels, level members, dimension hierarchies, and the association of aggregate functions to measures without affecting the existing observations [62] or using QB4OLAP from scratch [22,29,44].

In [36], the authors present an approach to enable OLAP operations on a single data cube published using the QB vocabulary and shown the applicability of their OLAP-to-SPARQL mapping in answering business questions in the financial domain. However, their OLAP-to-SPARQL mapping may not result in
the most efficient SPARQL query and requires additional efforts and a longer time to get the results as they consider that the cube is queried on demand and the DW is not materialized. While some approaches have proposed techniques to optimize execution of OLAP-style SPARQL queries in a federated setting [30], others have considered view materialization [21,31]. The authors in [62] present a semi-automatic method to enrich the QB dataset with QB4OLAP terms. However, there is no guideline for an ETL process to populate a DW annotated with QB4OLAP terms.

After analyzing the two research lines, we can draw some conclusions. Although each approach described above addresses one or more aspects of a semantic ETL framework, there is no single platform that supports them all (target definition, source-to-target mappings generation, ETL generations, MD target population, and evolution). To solve this problem, we have proposed a Python-based programmable semantic ETL (SETLPROG) framework [44] that provides a number of powerful modules, classes, and methods for performing the tasks mentioned above. It facilitates developers by providing a higher abstraction level that lowers the entry barriers. We have experimentally shown that SETLPROG performs better in terms of programmer productivity, knowledge base quality, and performance, compared to other existing solutions. However, to use it, developers need a programming background. Although SETLPROG enables to create an ETL flow and provides methods by combining several tasks, there is a lack of a well-defined set of basic semantic ETL constructs that allow users more control in creating their ETL process. Moreover, how to update an SDW to synchronize it with the changes taking place in the sources is not discussed. Further, in a data lake/big data environment, the data may come from heterogeneous formats, and the use of the relational model as the canonical model may generate an overhead. Transforming JSON or XML data to relational data to finally generate RDF can be avoided by using RDF as the canonical model instead. To this end, several works have discussed the appropriateness of knowledge graphs for data integration purposes and specifically, as a canonical data model [14,54,55]. An additional benefit of using RDF as a canonical model is that it allows adding semantics without being compliant to a fixed schema. The present paper presents the improvements introduced on top of SETLPROG to remedy its main drawbacks discussed above. As there are available RDF Wrappers (e.g., [18,57]) to convert another format to RDF, in this paper, we focus on only semantic data and propose an RDF-based two-layered (Definition Layer and Execution Layer) integration process. We also propose a set of high-level ETL constructs (tasks/operations) for each layer, with the aim of overcoming the drawbacks identified above for SETLPROG. We also provide an operation to update an SDW based on the changes in source data. On top of that, we characterize the ETL flow in the Definition Layer by means of creating an RDF based source-to-target mapping file, which allows to automate the ETL execution flows.

10. Conclusion and future work

In this paper, we proposed a framework of a set of high-level ETL constructs to integrate semantic data sources. The overall integration process uses the RDF model as canonical model and is divided into the Definition and Execution Layer. In the Definition Layer, ETL developers create the metadata (target and source TBoxes, and source-to-target mappings). We propose a set of high-level ETL operations for semantic data that can be used to create ETL data flows in the Execution Layer. As we characterize the transformations in the Definition Layer in terms of source-to-target mappings at the schema level, we are able to propose an automatic algorithm to generate ETL data flows in the Execution Layer. We developed an end-to-end prototype SETLCONSTRUCT based on the high-level constructs proposed in this paper. We also extended it to enable automatic ETL execution flows generation (and named it SETLAUTO). The experiment shows that 1) SETLCONSTRUCT uses 92% fewer NOTC than SETLPROG, and SETLAUTO further reduces NOUC by another 25%; 2) using SETLCONSTRUCT, the development time is almost cut in half compared to SETLPROG, and is cut by another 27% using SETLAUTO; 3) SETLCONSTRUCT is scalable and has similar performance compared to SETLPROG.

SETLCONSTRUCT allows users to create source-to-target mappings manually. However, an extension of this work is to create the mappings (semi-)automatically. Although this kind of ETL mappings is different from traditional ontology schema matching techniques, we can explore those techniques to establish the relationships among the concepts across the source and target ontologies. Taking the output of this schema matching approach as a suggestion, the expert can enrich the mappings according to the business requirements [41]. As the correctness of an ontology schema
matching technique depends on the semantic-richness of the given ontologies, besides the extensional knowledge given in the ABox, we need to take other external standard ontologies into account in the process of the \textit{TBoxExtraction} operation.

The source-to-target mappings act as a mediator to generate data according to the semantics encoded in the target. Therefore, we plan to extend our framework from purely physical to also virtual data integration where instead of materializing all source data in the DW, ETL processes will run on demand. When considering virtual data integration, it is important to develop query optimization techniques for OLAP queries considering virtual data integration, it is important to develop query optimization techniques for OLAP queries.

A.1. Semantics of the Execution Layer operations

In this section, we provide the detailed semantics of the ETL operations described in Section 6. The operations depend on some auxiliary functions. Here, we first present the semantics of the auxiliary functions and then the semantics of the operations. We present each function and operation in terms of input Parameters and semantics. To distinguish the auxiliary functions from the operations, we use small capital letters to write an auxiliary function name, while an operation name is written in italics.

A.1.1. Auxiliary functions

\textbf{EXECUTEQUERY}(Q, G, outputHeader) This function provides similar functionality as SPARQL \textbf{SELECT} queries.

\textbf{Input Parameters:} Let $I$, $B$, $L$, and $V$ be the sets of IRIs, blank nodes, literals, and query variables. We denote the set of RDF terms $(I \cup B \cup L)$ as $T$ for an RDF graph $G$. $V$ is disjoint from $T$. A query variable $v \in V$ is prefixed by the symbol '?'. A query pattern $Q$ can be recursively defined as follows:\footnote{We follow the same syntax and semantics used in [50] for a query pattern.}

1. An RDF triple pattern $t_p$ is a query pattern $Q$. A $t_p$\footnote{To make it easily distinguishable, here we use comma to separate the components of a triple pattern and an RDF triple.} allows query variables in any position of an RDF triple, i.e., $t_p \in (I \cup B \cup V) \times (I \cup V) \times (T \cup V)$.
2. If $Q_1$ and $Q_2$ are query patterns, then $(Q_1 \text{ AND } Q_2)$, $(Q_1 \text{ OPT } Q_2)$, and $(Q_1 \text{ UNION } Q_2)$ are also query patterns.
3. If $Q$ is a query pattern and $F_c$ is a filter condition then $(Q \text{ FILTER } F_c)$ is a query pattern. A filter condition is constructed using elements of the set ($T \cup V$) and constants, the equality symbol ($=$), inequality symbols ($<$, $\geq$, $\leq$, $>$), logical connectivities ($\neg$, $\vee$, $\land$), unary predicates like bound, isBlank, and isIRI plus other features described in [25].

$G$ is an RDF graph over which $Q$ is evaluated and \textit{outputHeader} is an output header list, which is a list of query variables and/or expressions over the variables used in $Q$. Here, expressions are standard SPARQL expressions defined in [25].

\textbf{Semantics:} For the query pattern $Q$, let $\mu$ be a partial function that maps $\text{var}(Q)$ to $T$, i.e., $\mu : \text{var}(Q) \rightarrow T$. The domain of $\mu$, denoted by $\text{dom}(\mu)$, is the set of variables occurring in $Q$ for which $\mu$ is defined. We abuse notation and say $\mu(Q)$ is the set of triples obtained by replacing the variables in the triple patterns of $Q$ according to $\mu$. The semantics of the query pattern are recursively described below.

1. If the query pattern $Q$ is a triple pattern $t_p$, then the evaluation of $Q$ against $G$ is the set of all mappings that can map $t_p$ to a triple contained in $G$, i.e., $[Q]_G = [t_p]_G = \{\mu \mid \text{dom}(\mu) = \text{var}(t_p) \land \mu(t_p) \in G\}$.
2. If $Q$ is $(Q_1 \text{ AND } Q_2)$, then $[Q]_G$ is the natural join of $[Q_1]_G$ and $[Q_2]_G$, i.e., $[Q]_G = [Q_1]_G \bowtie [Q_2]_G = \{\mu_1 \cup \mu_2 | \mu_1 \in [Q_1]_G, \mu_2 \in [Q_2]_G\}$, where $\mu_1$ and $\mu_2$ are compatible mappings, i.e., $\forall \pi \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2), \mu_1(\pi) = \mu_2(\pi)$.

3. If $Q$ is $(Q_1 \text{ OPT } Q_2)$, then $[Q]_G$ is the left outer join of $[Q_1]_G$ and $[Q_2]_G$, i.e., $[Q]_G = [Q_1]_G \triangleright \bowtie [Q_2]_G = ([Q_1]_G \bowtie [Q_2]_G) \cup ([Q_1]_G \setminus ([Q_2]_G \setminus [Q_2]_G))$.

4. If $Q$ is $(Q_1 \text{ UNION } Q_2)$, then $[Q]_G = [Q_1]_G \cup [Q_2]_G = \{\mu | \mu \in [Q_1]_G \text{ or } \mu \in [Q_2]_G\}$.

5. If $Q = (Q_1 \text{ FILTER } F_c)$, then $[Q]_G = \{\mu | \mu \in [Q_1]_G \land \mu \models F_c\}$, where a mapping $\mu$ satisfies a filter condition $F_c$, denoted by $\mu \models F_c$.

This function returns a set of tuples where each tuple contains the values of the variables and/or the values derived by solving the expressions defined in outputHeader according to $\mu$.

\textsc{getPropertiesFromExpressions}(sTBox, exps)

An expression is a combination of one or more properties, operators, and SPARQL functions defined in [25]. This function returns a subset of properties of a source TBox that are used in the given expressions.

\textbf{Input Parameters:} sTBox is a TBox and exps is a set of expressions.

\textbf{Semantics:} For each $exp \in \text{exps}$, this function computes the intersection of the set of IRI used in $exp$ and the set of properties of sTBox. Then, it returns the union of all intersection sets. Here, returnIRIs(exp) returns the set of IRI used in $exp$ and $P(t)$ is defined in Equation (2). The semantic is defined as follows:

\textsc{getPropertiesFromExpressions}(sTBox, exps)

\[= \bigcup_{\text{exps} \in \text{exps}} \text{returnIRIs}(\text{exp}) \cap P(s\text{TBox}).\]

This function returns a set of properties.

\textsc{validateExpressions}(exps, Q, flag)

The source expressions/properties (defined by map:source4TargetPropertyValue) in property-mappings contains properties. However, the outputHeader parameter of \textsc{executeQuery()} allows only query variables. Therefore to extract data from an RDF graph using \textsc{executeQuery()}, the properties used in the expressions/properties of property-mappings should be replaced by corresponding query variables. On the other hand, to match the expressions/properties used in outputHeader of \textsc{executeQuery()} with the expressions/properties (defined by map:source4TargetPropertyValue) in property-mappings, the query variables used in expressions/properties need to be replaced by the corresponding properties. Given a set of expressions, a query pattern, and a flag, this function replaces the used term (either IRI or query variables) in expressions with alternative ones from the query pattern based on the flag and returns a list of the validated expressions.

\textbf{Input Parameters:} exps is a list of expressions, $Q$ is a query pattern, and flag indicates whether it will replace the IRIs (flag = 1) or query variables (flag = 0) used in $exps$.

\textbf{Semantics:} If flag = 1 the function replaces the $exps$ IRIs with the corresponding query variables used in $Q$. For each IRI $iri$ used in an expression, \textsc{validateExpressions} replaces $iri$ with the object of the triple pattern whose predicate is $iri$ in $Q$. Finally, it returns the list of expressions, where each expression does not contain any IRIs.

If flag = 0 the function replaces the $exps$ query variables with the corresponding predicate IRIs used in $Q$. For each query variable $?q$ used in an expression, it replaces $?q$ with the predicate of the triple pattern whose object is $?q$ in $Q$. Finally, it returns the list of expressions, where each expression does not contain any query variables.

\textsc{mappedSourceInstances}(sc, sTBox, sABox, map:source4TargetPropertyValue)

\textbf{This function returns a dictionary describing the instances of a source concept.}

\textbf{Input Parameters:} sc is a source construct, sTBox and sABox are the source TBox and ABox, propertyMappings is a set of property-mappings.

\textbf{Semantics:} At first, this function retrieves instances of sc with their corresponding properties and values. Here, we consider only those properties that are directly mapped to target properties and/or used in source expressions. Both the properties and expressions are defined in propertyMappings by map:source4TargetPropertyValue. Then, it creates a dictionary, a set of $(key, value)$ pairs. In a pair, key represents an instance IRI and value in turn represents a set of $(p_i, v_i)$ pairs, where $p_i$ represents a property and $v_i$ represents a value for $p_i$. It is explained as
follows:

MAPPEDSOURCEINSTANCES(sc, sTBox, sABox, propertyMapping) = dictionary(executeQuery((?i, rdf:type sc) AND (?i, ?p, ?v) FILTER (?p ∈ GETPROPERTIESFROMEXPRESSIONS(sTBox, mappedExpressions(sc, propertyMappings))), sABox, (?i, ?p, ?v))).

Here, mappedExpressions(sc, propertyMappings) returns the set of source properties/expressions (defined by map:source4TargetPropertyValue) used in propertyMappings. The dictionary((?i, ?p, ?v)) function first groups the input tuples by ?i and then for each instance i ∈ ?i, it creates a set of (p1, v1) pairs, where p1 is a property of i and v1 is the value for p1. Finally, MAPPEDSOURCEINSTANCES(…) returns the dictionary created.

GENERATEIRI(sIRI, value, tType, tTBox, iriGraph)
Every resource in an SDW is uniquely identified by an IRI defined under the namespace of the SDW. This function creates an equivalent target IRI for a source resource. Additionally, it keeps that information in the IRI graph.

Input Parameters: sIRI is the source IRI, value is the literal to be used to ensure the uniqueness of the generated IRI, tType is the type of the generated IRI in tTBox, tTBox is the target TBox, and iriGraph is the IRI graph. The IRI graph is an RDF graph that keeps a triple for each resource in the SDW with their corresponding source IRI.

Semantics: Equation (3) formulates how to create the IRI. First, the function checks whether there is an equivalent IRI for sIRI in iriGraph using lookup(sIRI, iriGraph). It returns the target IRI if it finds an existing IRI in iriGraph; otherwise, it generates a new IRI by concatenating prefix(tTBox) and validate(value) for a target concept or property, or creates an instance IRI by concatenating tType and validate(value). Here, prefix(tTBox) returns the namespace of tTBox; concat() concatenates the input strings; validate(value) modifies value according to the naming convention rules of IRIs described in [53]; C(T) and P(T) are defined in Equation (1) and (2). Upon the creation of the IRI, this function adds an RDF triple (tIRI owl:sameAs sIRI) to iriGraph.

generateIRI(sIRI, value, tType, tTBox, iriGraph)

\[
\begin{align*}
\text{lookup}(sIRI, \text{iriGraph}) & = \text{NULL} \\
\text{concat}(\text{tType}, \#) & = \text{value} \\
\text{validate}(\text{value}) & = \text{NULL} \land \text{tType} \in \{C(T) \cup \text{P}(T) \}
\end{align*}
\]

This function returns an IRI.

TUPLES_TO_TRIPLES(T, tc, Q, propertyMappings, (i, exp1, ..., expn)) As the canonical model of our integration process is RDF model, we need to convert the instance descriptions given in the tuple format into equivalent RDF triples. This function returns a set of RDF triples from the given set of tuples.

Input Parameters: T is a set of tuples (1st normal form tabular format), tc is a target construct, Q is a query pattern, propertyMappings is a set of property-mappings, and (i, exp1, ..., expn) is the tuple format.

Semantics: Equation (4) shows the semantics of this function. Each tuple of T represents an instance-to-be of the target construct tc, and each expression (expp) and the value of the expression (val(exp)) represent the corresponding target property and its value of the instance. Here, val(exp) is the value of an expression in a source tuple. As an expression can contain query variables and expressions used in propertyMappings do not use query variables, VALIDATE_EXPRESSIONS(exp, Q, 0) replaces the query variables with equivalent properties from Q, and return(x, propertyMappings) returns the target construct mapped to source construct x from propertyMappings.

TUPLES_TO_TRIPLES(T, tc, Q, propertyMappings, (i, exp1, ..., expn))

\[
\bigcup_{(i, \text{val}(\text{exp1}), ..., \text{val}(\text{expn})) \in T} \{(\text{val}(i), \text{rdf:type, tc})\} 
\]

\[
\bigcup_{p=1}^{n} \{(\text{val}(i), \text{return} (\text{VALIDATE_EXPRESSIONS}\exp_p, Q, 0))\}
\]

---

20If the source construct is either a QB level property or dataset, then rdf:type is replaced by either qb4o:memberOf or qb4o:dataset.
This function returns a set of RDF triples.

A.1.2. Execution Layer operations

This section gives the semantics of each operation category-wise.

Extraction Operations

GraphExtractor\((Q, G, outputPattern, tABox)\) This operation is functionally equivalent to SPARQL CONSTRUCT queries to extract data from sources.

Input Parameters: \( Q \) is a query pattern as defined in EXECUTE QUERY(). \( G \) is an RDF graph over which \( Q \) is evaluated. The parameter outputPattern is a set of triple patterns from \((I \cup B \cup \text{var}(Q)) \times (I \cup \text{var}(Q)) \times (T \cup \text{var}(Q))\), where \( \text{var}(Q) \subseteq V \) is the set of query variables occurring in \( Q \). Here, \( I, B, T, \) and \( V \) are the sets defined in EXECUTE QUERY(). The parameter tABox is the location where the output of this operation is stored.

Semantics: This operation generates the output in two steps: 1) First, \( Q \) is matched against \( G \) to obtain a set of bindings for the variables in \( Q \) as discussed in EXECUTEQUERY(). 2) Then, for each variable binding \( \mu \), it instantiates the triple patterns in outputPattern to create new RDF triples. The result of the query is a merged graph, including all the created triples for all variable bindings, which will be stored in tABox. Equation (5) formally defines it.

\[
\begin{align*}
\text{GraphExtractor}(Q, G, outputPattern, tABox) & = \bigcup_{t_h \in outputPattern} \{ \mu(t_h) | \mu \in \mathbb{Q}_G \land \mu(t_h) \text{ is a well-formed RDF triple} \}.
\end{align*}
\]

Transformation Operations

TransformationOnLiteral(sConstruct, tConstruct, sTBox, sABox, propertyMappings, tABox) This operation creates a target ABox from a source ABox based on the expressions defined in property-mappings.

Input Parameters: \( s\text{Construct} \) and \( t\text{Construct} \) are a source and target TBox construct, \( s\text{TBox} \) and \( s\text{ABox} \) are the source TBox and ABox, \( \text{propertyMappings} \) is a set of property-mappings, and \( t\text{ABox} \) is the output location.

Semantics: First, we retrieve the instances of \( s\text{Construct} \) from \( s\text{ABox} \) using the EXECUTEQUERY() function, which is formally described as follows: \( \text{ins}(c) = \text{EXECUTEQUERY}(q(c), s\text{ABox}, (?i, \text{VALIDATEEXPRESSIONS}(\text{list}(c\text{Elements}), q(c), 1)) \) .

Here:

- \( c = s\text{Construct} \).
- \( q(c) = \text{((?i, \text{rdf:type}, c) \text{ AND } (?i, ?p, ?v) \text{ FILTER } (?p \in c\text{Properties}))} \) is a query pattern.
- \( c\text{Properties} = \text{GETPROPERTIESFROMEXPRESSIONS}(s\text{TBox}, c\text{Elements}) \) is the set of source properties used in source expressions defined in propertyMaps.
- \( c\text{Elements} = \text{EXECUTEQUERY}((?pm, \text{map}: \text{source4TargetPropertyValue}, ?sp), \text{propertyMaps}, (?sp) \) is the set of source expressions in propertyMaps defined by map: source4TargetPropertyValue.
- \( \text{VALIDATEEXPRESSIONS}(\text{list}(c\text{Elements}), q(c), 1) \) replaces the source properties used in cElements with the corresponding query variables from q(c) as outputHeader parameter EXECUTEQUERY() does not allow any properties. Since VALIDATEEXPRESSIONS takes a list of expressions as a parameter, list(cElement) creates a list for scElements.

Now, we transform all tuples of \( \text{ins}(c) \) into equivalent RDF triples to get \( t\text{ABox} \), i.e.,

\[
\text{output} = \text{TUPLESToTRIPLES}((\text{ins}(c), \text{tConstruct}, q(c), \text{(?i, exp1, \ldots, expn)})).
\]

The output of this operation is output, a set of RDF triples, which will be stored in tABox.

JoinTransformation(sConstruct, tConstruct, sTBox, tTBox, sABox, tABox, comProp, propertyMappings) This operation joins and transforms the instances of source and target based on property-mappings.

Input Parameters: \( s\text{Construct} \) and \( t\text{Construct} \) are a source and target \( T\text{Box} \) construct, \( s\text{TBox} \) and \( s\text{ABox} \) are the source and target \( T\text{Boxes} \); \( s\text{ABox} \) and \( t\text{ABox} \) are the source and target ABoxes; \( \text{comProp} \) is a set

\footnote{The term target here does not mean target schema but in the sense it is defined in concept-mappings. Source and target both can be either source concepts or intermediate concepts (from an intermediate result).}
of common properties between tConstruct and sConstruct and propertyMappings is the set of property-mappings.

**Semantics:** At first, we create an ABox by taking the set union of instances (including their properties and values) of both sConstruct and tConstruct and apply a query on the ABox using the EXECUTEQUERY() function. The query pattern of the function joins the instances of two concepts based on their common properties and finally the function returns a set of target instances ins(sc,tc) with the values of the source expressions (defined by map:source4TargetPropertyValue) in propertyMappings. As the query pattern includes triple patterns from both sConstruct and tConstruct, the source expression can be composed of both source and target properties. This is described as follows:

\[
\text{ins(tc, sc)} = \text{EXECUTEQUERY}((q(tc)\text{OPTq(sc)}),
\]

\[
\quad \text{union(extractIns(tc, tABox), extractIns(sc, sABox)), (?i, VALIDATEEXPRESSIONS(scElements, (q(tc)\text{OPTq(sc)}), 1)).}
\]

Here:

- \(tc = t\text{Construct}\) and \(sc = s\text{Construct}\).
- \(q(tc) = ((?\text{tc}, \text{rdf:type}, tc) \text{ AND } (?i_{\text{tc}}, ?p_{\text{tc}}, ?v_{\text{tc}}) p_{\text{con}}(\text{"target"}, \text{comProp}) \text{ FILTER}(p_{\text{tc}} \in tc\text{Properties}))\) is a query pattern.
- \(p_{\text{con}}(\text{"target"}, \text{comProp})\) is a function that joins a triple pattern (i.e., AND (\(?i_{\text{sc}}, \text{comProp}\), \(?p_{\text{sc}}, \text{comProp}\)) in \(q(tc)\) for each pair (\(\text{comProp}\)) in \(tc\text{Properties}\).
- \(\text{scProperties} = \text{GETPROPERTIESFROMEXPRESSIONS(sTBox, scElements)}\) represents the set of source properties used in source expressions.
- \(scElements = \text{EXECUTEQUERY}((?p_{\text{m}}, \text{map:source4TargetPropertyValue}, \text{?sp}), \text{propertyMappings}, \text{?sp})\) is the set of source expressions in propertyMappings defined by map:source4TargetPropertyValue.
- \(\text{extractInstance}(c, abox) = \text{EXECUTEQUERY}(((?i, \text{rdf:type}, c) \text{ AND } (?i, ?p, ?v)), abox,(?i, ?p, ?v))\) retrieves the instances (with their linked properties and values) of the concept \(c\) from the given ABox \(abox\).
- \(\text{union}(s1, s2)\) returns the set union of two given sets \(s1\) and \(s2\).
- \(\text{VALIDATEEXPRESSIONS}(\text{list(scElements)}, q(sc, tc), I)\) replaces the source properties used in \(sc\text{Elements}\) with the corresponding query variables from \(q(tc,sc)\) as the outputHeader parameter of EXECUTEQUERY() does not allow any properties. Since \(\text{VALIDATEEXPRESSIONS}\) takes a list of expressions as a parameter, list(cElement) creates a list for \(sc\text{Elements}\).

Now, we transform all tuples of \(\text{ins(sc, tc)}\) into equivalent RDF triples to get the transformed \(tABox\), i.e.,

\[
\text{output} = \text{TUPLES TO TRIPLES}(\text{ins(tc, sc), tConstruct}, q(tc, sc), (\text{?i}, \text{exp1}, \ldots, \text{exp}_n)).
\]

The output of this operation is \(\text{output}\), a set of RDF triples, which will be stored in \(tABox\).

**LevelMemberGenerator(sConstruct, level, sTBox, sABox, iTBox, iriValue, iriGraph, propertyMappings, tABox)** The objective of this operation is to create QB-OLAP-compliant level members.

**Input Parameters:** \(s\text{Construct}\) is the source construct, \(level\) is a target level, \(s\text{TBox}\) and \(s\text{ABox}\) are the source TBox and ABox, \(iriValue\) is the rule of creating level members’ IRIs, \(iriGraph\) is the IRI graph, \(propertyMappings\) is a set of property-mappings, and \(tABox\) is the output location.

**Semantics:** First, we retrieve the instances of \(s\text{Construct}\) with their properties and values, i.e.,

\[
\text{Ins} = \text{MAPPEDSOURCEINSTANCES(sConstruct, sTBox, sABox, propertyMappings)}
\]  

\(6\)
To enrich level with the dictionary \textit{Ins}, we define a set of triples \textit{LM} by taking the union of \textit{IdentityTriples} (for each \((i, pv)\) pair in \textit{Ins}) and the union of \textit{DescriptionTriples} (for each \((p, v, i)\) pair in \textit{pv}). Equation (7) defines \textit{LM}.

\[
\text{LM} = \bigcup_{(i, pv) \in \text{Ins}} \left( \text{IdentityTriples} \cup \bigcup_{(p, v, i) \in \text{pv}} \text{DescriptionTriple} \right)
\]

(7)

Here:

- \textit{IdentityTriples}

\[
\text{IdentityTriples} = \{(\text{lmIRI}, \text{rdf}\text{-type}, \text{qb4o}\text{:LevelMember}), (\text{lmIRI}, \text{qb4o}\text{:memberOf}, \text{level})\}
\]

(8)

- \textit{lmIRI}

\[
\begin{align*}
\text{lmIRI} &= \text{GENERATEIRI}(i, \text{resolve}(i), pv, \text{iriValue}), \text{range}((\text{level}, \text{tTBox}, \text{iriGraph}), \text{tBox}) \\
&= \begin{cases} \\
\text{GENERATEIRI}(i, \text{resolve}(i), pv, \text{iriValue}), \text{range}((\text{level}, \text{tTBox}, \text{iriGraph}), \text{tBox}) & \text{if range(level, tBox)} = \text{NULL} \\
\text{GENERATEIRI}(i, \text{resolve}(i), pv, \text{iriValue}), \text{level}, \text{tTBox}, \text{iriGraph}) & \text{if range(level, tBox)} = \text{NULL} \\
\end{cases}
\end{align*}
\]

(9)

- \textit{DescriptionTriple}

\[
\begin{align*}
\text{DescriptionTriple} &= \{(\text{lmIRI}, \text{return}(p, \text{propertyMappings}), \text{GENERATEIRI}(v, \text{iriValue}(v), \text{range}(\text{return}, v, \text{propertyMapping})), (\text{tBox}, \text{tBox}, \text{iriGraph})) & \text{if targetType}(\text{return}, v, \text{propertyMappings}, \text{tBox}) \in \{\rule{0pt}{0pt}\text{nullProperty, ObjectProperty}\} \\
&= \{(\text{lmIRI}, \text{return}(p, \text{propertyMappings}, v), \text{return}(\text{propertyMapping}, \text{tBox}), \text{levelAttribute}) & \text{if targetType}(p, \text{propertyMappings}, \text{tBox}) = \text{levelAttribute} \}
\end{align*}
\]

(10)

Each instance of \textit{sConstruct} is a \textit{qb4o}\text{:LevelMember} and a member of \textit{level}; therefore, for each instance of \textit{sConstruct}, \textit{LM} in Equation (7) includes two identity triples, described in Equation (8). Equation (9) describes how to create an IRI for a level member. As we described in Section 5, the rule for creating IRIs can be different types. If \textit{iriValue} is “sameAsSourceIRI” then the function returns the source IRI; otherwise it resolves value of \textit{iriValue} using the resolve\((i, pv, \text{iriValue})\) function – this function returns either the value of a property/expression or next incremental value \(\text{nullIRI}\), and finally it creates a new IRI by calling the GENERATEIRI() function. As some datasets (e.g., [22]) use the IRI of the level to create the IRIs of its members, whereas others (Eurostat (https://ec.europa.eu/eurostat/data/database), Greece (http://linked-statistics.gr/) linked datasets) use the IRI of the range of the level, we generalize it using Equation (9).

If a range exists for the target level, the IRI of that range is used as a target type – the type of the resources for which IRIs will be created – for creating IRIs for the members of that level (second case in Equation (9)); otherwise, the level’s IRI is used as a target type (third case in Equation (9)). Here, range\((\text{level}, \text{tBox})\) returns the range of level from \textit{tBox}.

Equation (10) creates a triple for each \((p, v, i)\) pair of \textit{Ins} (in Equation (6)). If \(p_i\) corresponds to either an object property or a rollup property in \textit{tBox}, then a new IRI is created for \(v_i\) and the target type of the IRI is the range of the target construct that is mapped to \(p_i\) (first case in Equation (10)). Here, \textit{return}(\(x\), mappings) function returns the target construct that is mapped to \(x\) from the set of property-mappings mappings; targetType\((x, \text{tBox})\) returns the type of a target construct \(x\) from \textit{tBox} and \textit{iriValue}(\(v\)) retrieves the value to be used to create the IRI. If \(v\) is a literal, it simply returns it, otherwise, it splits \(v\) by either “/” or “#” and returns the last portion of \(v\). If \(p_i\) corresponds to a level attribute (i.e., datatype property), then the object of the triple generated for the level member will be the same as \(v\) (second case in Equation (10)).

The output of this operation is \textit{LM} and the operation stores the output in \textit{tBox}.

\textit{ObservationGenerator}(\textit{sConstruct, dataset, tBox, sABox, tBox, iriValue, iriGraph, propertyMappings, tBox}) This operation creates QB4OLAP-compliant observations from the source data.

\textbf{Input Parameters:} \textit{sConstruct} is the source construct, \textit{dataset} is a target QB dataset, \textit{sBox} and \textit{sABox} are the source TBox and ABox, \textit{iriValue} is the rule of creating level members’ IRIs, \textit{iriGraph} is the IRI graph, \textit{propertyMappings} is a set of property-mappings, and \textit{tBox} is the output location.

\textbf{Semantics:} First, we retrieve the instances of \textit{sConstruct} with their properties and values using Equation (6). To populate \textit{dataset} with the dictionary \textit{Ins}, we define a set of triples \textit{OB} which is equivalent to \textit{LM}
in Equation (7).

\[ \text{IdentityTriples} = \{(oIRI, \text{rdf:type}, \text{qb:Observation}), (oIRI, \text{qb:dataset}, \text{dataset})\} \]  \hspace{1cm} (11)

\[ \text{oIRI} = \begin{cases} i & \text{if iriValue = "sameAsSourceIRI"} \\ \text{generateIRI}(i, \text{resolve}(vi), \text{propertyMappings}, \text{tBox}, \text{iriGraph}) & \text{otherwise} \end{cases} \]  \hspace{1cm} (12)

\[ \text{DescriptionTriple} = \begin{cases} \{(oIRI, \text{return}(p_1, \text{propertyMappings}, \text{lmIRI})), \text{tTBox} \} & \text{if targetType} = \text{LevelProperty} \\ \{(oIRI, \text{return}(p_1, \text{propertyMappings}, v_i)), \text{tTBox} \} & \text{if targetType} = \text{MeasureProperty} \end{cases} \]  \hspace{1cm} (13)

\[ \text{lmIRI}_o = \begin{cases} \text{generateIRI}(v_i, \text{iriValue}(v_i), \text{range} \text{return}(p_1, \text{propertyMappings}), \text{tBox}, \text{iriGraph}) & \text{if range} \text{return}(p_1, \text{propertyMappings}, \text{tBox}) \text{NULL} \\ \text{generateIRI}(v_i, \text{iriValue}(v_i), \text{return}(p_1, \text{propertyMappings}), \text{tBox}, \text{iriGraph}) & \text{otherwise} \end{cases} \]  \hspace{1cm} (14)

Each instance of \text{sConstruct} is a \text{qb:Observation} and the QB dataset of that observation is \text{dataset}; therefore, for each instance of \text{sConstruct}, \text{OB} in Equation (7) includes two identity triples. Equation (11) redefines Equation (8) for \text{OB}. Equation (12) describes how to create an \text{IRI} for an observation. If \text{iriValue} is “sameAsSourceIRI” then the function returns the source \text{IRI}; otherwise it generates an observation \text{IRI} by calling the \text{generateIRI()} function. Equation (13) redefines Equation (10) for \text{OB}. For each \((p_1, v_i)\) in the dictionary, the equation creates a triple. If \(p_1\) represents a level in the target, the predicate of the triple is the corresponding level of \(p_1\) and the object of the triple is a level member of that level. Equation (14) shows how to create the \text{IRI} for the object (level member) of that triple. If \(p_1\) represents a measure property in the target, the predicate of the triple is the corresponding target measure property of \(\text{tBox}\) and the object of the target triple is similar to \(v_i\).

The output of this operation is \text{OB}, which will be stored in \text{tABox}.

\text{ChangedDataCapture(nABox, oABox, flag)} This operation triggers either the SDW evolution (i.e., enriched with new instances) or update (i.e., reflect the changes in the existing instances).

\text{Input Parameters:} \text{nABox} and \text{oABox} are the sets of triples that represent the new source data and the old source data. \text{flag} indicates the type of difference the operation will produce. If \text{flag} = 0, the operation produces the new instances, and if \text{flag} = 1, the operation produces the set of updated triples (i.e., the triples in \text{oABox} updated in \text{nABox}).

\text{Semantics:} Here, \text{nABox} and \text{oABox} both are a set of triples where first element of each triple corresponds to a level member in the target, second element represents a property (either a level attribute or a rollup property) and the third element represents a value of the property for that instance. First, we retrieve the sets of instances from \text{nABox} and \text{oABox} using the \text{EXECUTE QUERY()} function, shown as follows:

\text{Ins}_{nABox} = \text{EXECUTE QUERY}((?i, \text{rdf:type}, ?v), \text{nABox}, ?i)

\text{Ins}_{oABox} = \text{EXECUTE QUERY}((?i, \text{rdf:type}, ?v), \text{oABox}, ?i)

The set difference of \text{Ins}_{nABox} and \text{Ins}_{oABox} gets the new instances, i.e., \text{Ins}_{new} = \text{Ins}_{nABox} \ominus \text{Ins}_{oABox}. We get the description (properties and their values) of new instances using Equation (15).

\[ \text{Ins}_{new} = \bigcup_{i\in \text{Ins}_{new}} \text{EXECUTE QUERY}((?s, ?p, ?v) \land \text{FILTER}(?s = i), \text{nABox}, (?s, ?p, ?v)) \]  \hspace{1cm} (15)

To get the triples that are changed over time, we first remove the description of new instances (derived in Eq. (15)) from \text{newABox} i.e., \text{Ins}_{new} = \text{newABox} \ominus \text{Ins}_{new}; then the set of changed triples by taking the set difference of \text{Ins}_{old} and \text{oldABox}, i.e.,

\text{ChangedTriples} = \text{Ins}_{old} \ominus \text{oldABox}.

If \text{flag} = 0 then the output is \text{Ins}_{new}, else \text{ChangedTriples}. The output overwrites \text{oABox}.

\text{updateLevel(level, updatedTriples, sABox, tBox, \text{tABox}, \text{propertyMappings}, \text{iriGraph})} This operation reflects the changes in source to the SDW.
**Input Parameters:** level is a target level, updatedTriples is the set of updated triples (generated by the ChangedDataCapture operation), sABox is the source data of level, tBox and tABox are the target TBox and ABox, propertyMappings is the set of property-mappings, and iriGraph is the IRI graph.

**Semantics:** As we consider that the changes only occur at the instance level, not at the schema level, only the 3rd elements (objects) of the triples of sABox and updatedTriples can be different. For the simplicity of calculation, we define sABox, and updatedTriples as the following relations:

\[ sABox = (\text{instance}, \text{property}, \text{oldVal}) \]

\[ \text{updatedTriples} = (\text{instance}, \text{property}, \text{newVal}) \]

This operation updates tBox by deleting invalid triples from tABox and inserting new triples that reflect the changes. As SPARQL does not support any (SQL-like) update statement, we define two sets DeleteTriples and InsertTriples that contain the triples to be deleted from tABox and inserted to tABox, respectively. To get the final tABox, we first take the set difference between tABox and DeleteTriples using Equation (16) and then, take the union of tABox and InsertTriples using Equation (17).

\[ \text{tABox} = \text{tABox} - \text{DeleteTriples} \]  
\[ \text{tABox} = \text{tABox} \cup \text{InsertTriples} \]

To get the updated objects and old objects of triples for source instances, we take the natural join between updatedTriples and sABox, i.e.,

\[ \text{NewOldValues} = \text{updatedTriples} \bowtie \text{sABox} \]

We define DeleteTriples and InsertTriples as

\[ \text{DeleteTriples} = \bigcup_{(i, p, nV) \in \text{updatedTriples}} \text{del}(i, p, nV) \]

\[ \text{InsertTriples} = \bigcup_{(i, p, nV, oV) \in \text{NewOldValues}} \text{in}(i, p, nV, oV) \]

The del\((i, p, nV)\) and in\((i, p, nV)\) functions depend on the update type of the level attribute corresponding to \(p\). We retrieve the update type of the level attribute from tBox using Equation (21). The updateType\((prop, tBox)\) function returns the update type of a level attribute \(prop\) from tBox (see the blue-colored rectangle of Fig. 1).

\[ \text{updateType} = (\text{updateType}(	ext{return}(p, \text{propertyMappings}), tBox)) \]  
\[ \text{EXECUTE QUERY} = \text{EXECUTE QUERY}((\text{?i, ?p, ?val}) \]

\[ \text{FILTER}((\text{?i = IRI,} \&\& \text{?p = return}(p, \text{propertyMappings}))) \]

Equation (23) describes how to create an RDF triple for the Type-1 update. First, we retrieve the equivalent value of \(oV\) in the target using Equation (24). As the retrieved value can be either a literal or an IRI, we create the object of the triple by replacing the \(oV\) portion of that value with \(nV\). The replace\((\text{org_str, search_pattern, replace_pattern})\) function updates the string \(\text{org_str}\) by replacing the substring \(\text{search_pattern}\) with \(\text{replace_pattern}\). The subject of the output RDF triple is the level member equivalent to \(i\) and the predicate is the target property equivalent to \(p\).

\[ \text{replace}(\text{targetValue}(i, p), \text{ov}) \]

\[ \text{in}(i, p, nV, oV) \]

\[ \text{del}(i, p, nV) \]

\[ \text{in}(i, p, nV, oV) \]

\[ \text{del}(i, p, nV) \]

\[ \text{in}(i, p, nV, oV) \]
where,

targetValue(i, p)

\[ = \text{EXECUTEQUERY}((\text{IRI}_i, \\text{return}(p, \text{mapping}), ?v), \text{tABox}, ?v) \]  

(24)

If updateType = Type-3 update

Like the Type-1 update, Type-3 update also replaces the old property value with the current one. Besides, it also keeps the old property value as the latest old property value. Therefore, the triples that contain the current and old property value should be removed from the target. Equation (25) defines del(i, p, nV) for the Type-3 update, which retrieves the triples containing both current and old property values.

As besides new value Type-3 keeps the latest old value of the property by adding another property, we define \( \text{in}(i, p, nV, oV) \) in Equation (26) which creates a triple for the new value and a triple for the old one. For a property \( \text{property} \), \( \text{concat}(\text{property}, \text{"_oldValue"}) \) creates a property to contain the old value of the property by concatenating the \( \text{"_oldValue"} \) with \( \text{property} \). The function \( \text{targetValue}(i, p) \) returns the objects of triple whose subject and predicate correspond to \( i \) and \( p \), defined in Equation (24).

\( \text{del}(i, p, nV) \)

\[ = \text{EXECUTEQUERY}((?i, ?p, ?val)
\]

\[ \quad \text{FILTER}(?i = \text{IRI}_i & & ?p \in (\text{return}(p, \text{propertyMappings}), \text{concat}(\text{return}(p, \text{propertyMappings}), \text{"_oldValue"})), \text{tABox}, (?i, ?p, ?val)) \]  

(25)

\( \text{in}(i, p, nV, oV) \)

\[ = \{ (\text{IRI}_i, \text{return}(p, \text{mapping}), \text{replace} (\text{targetValue}(i, p, oV, nV), \text{IRI}_i, \text{concat}(\text{return}(p, \text{propertyMappings}), \text{"_oldValue"}), \text{targetValue}(i, p))) \} \]  

(26)

If updateType = Type-2 update

In Type-2 update, a new version for the level member is created (i.e., it keeps the previous version and creates a new updated version). Since the validity interval (defined by \( \text{type2:toDate} \)) and status (defined by \( \text{type2:status} \)) of the previous version need to be updated, triples describing the validity interval and status of the previous version should be deleted from \( \text{tABox} \). Equation (27) defines \( \text{del}(i, p, nV) \) for Type-2 update. The first operand of the union in Equation (27) retrieves the expired triples of \( \text{IRI}_i \) (the level member corresponding to \( i \)) from \( \text{tABox} \). As the level of \( \text{IRI}_i \) can be an upper level in a hierarchy, the validity intervals and the status of the level members of lower levels referring to \( \text{IRI}_i \) need to be updated, too. Therefore, the current validity intervals and status of the associated level members will also be deleted. The second operand of union in Equation (27) \( \text{getExpiredTriplesAll} (\text{IRI}_i) \) returns the set of expired triples of the members of all lower levels referring to \( \text{IRI}_i \) (described in Equation (28)). The function \( \text{getTriplesImmediate} (\text{IRI}_i) \) returns the set of expired triples of the members of immediate child level referring to \( \text{IRI}_i \) (described in Equation (29)).

\( \text{del}(i, p, nV) \)

\[ = \text{EXECUTEQUERY}((\text{IRI}_i, ?p, ?val)
\]

\[ \quad \text{FILTER}(?p \in (\text{type2:toDate}, \text{type2:status})), \text{tABox}, (?i, ?p, ?val)) \]  

(27)

\( \text{getExpiredTriplesAll}(\text{IRI}_i) \)

\[ = \bigcup_{(i, p, v) \in \text{getExpiredTriplesImmediate} (\text{IRI}_i)} \text{getExpiredTriplesAll}(i, c) \]  

(28)

\( \text{getExpiredTriplesImmediate}(\text{IRI}_i) \)

\[ = \text{EXECUTEQUERY}((\text{i}, ?p, \text{IRI}_i) \text{ AND } (\text{i}, \text{rdf:type, qb4o:LevelMember}) \text{ AND } (?i, ?p, ?v) \text{ FILTER}(?p \in (\text{type2:toDate}, \text{type2:status})), \text{tABox}, (?i, ?p, ?val)) \]  

(29)

To reflect the changes in the target, a new version of the level member (with the changed and unchanged property values) is created. Equation (30) defines \( \text{in}(i, p, nV, oV) \) for Type-2 update. The IRI for the new version of \( \text{IRI}_i \) is \( \text{newIRI}_i = \text{updateIRI}(\text{IRI}_i, \text{iriGraph}) \). The \( \text{updateIRI}(\text{iri, iriGraph}) \) function updates the existing IRI \( \text{iri} \) by appending the current date with it and returns the updated one. An RDF triple is generated for \( nV \), where the object is composed by replac-
ing the old value with the new value, the predicate is the target level attribute equivalent to \( p \), and the subject is \( \text{newIRI} \), (first operand of the union operation in Equation (30)). The call of \( \text{update2Insert}(\text{iri}_o, \text{iri}_n, \text{cp}) \) in Equation (30) returns the triples required to reflect the changes in \( \text{tABox} \). Here, \( \text{iri}_i \) and \( \text{iri}_n \) are the old and new version (IRI) of a level member and \( \text{cp} \) is the predicate of the updated triple (i.e., the property whose value has been changed). The values of \( \text{type2:status} \) for new and existing versions are set to “Current” and “Expired”, respectively. The validity interval of existing version is ended on \( \text{sysdate}()-1 \) and the validity interval of new version is from \( \text{sysdate}() \) to “9999-12-31”. The \( \text{sysdate}() \) returns the current date in year-month-day format as we consider that at most one update operation can be applied on the SDW in a day. The value “9999-12-31” in the \( \text{type2:toDate} \) indicates that the instance is still valid; this is a usual notation in temporal databases [61]. As the new version of the level member also contains the triples for its unchanged properties, we retrieve the triples of a level member’s old version \( \text{iri}_o \) with unchanged properties using \( \text{retrieveTriples}(\text{iri}_o, \text{cp}) \) which returns all triples of \( \text{iri}_o \) except the triples with the properties \( \text{cp} \) (changed property), \( \text{type2:toDate} \), \( \text{type2:fromDate} \), and \( \text{type2:status} \) (described in Equation (32)). We replace the IRI of old version with the new one using the \( \text{replace()} \) function. To propagate the current version to its descendants (i.e., the level members those are associated to \( \text{iri}_o \)); the function \( \text{updateAssociates}(\text{iri}_o, \text{iri}_n) \) creates new versions for the associated members of all lower levels in the hierarchy so that they refer to the correct version (described in Eq. (33)). The function \( \text{getAssociates}(\text{iri}_o) \) returns the set of triples that describe the members that are connected to \( \text{iri}_o \) with a property. For each member connected to \( \text{iri}_o \), we create a new version of it and also recursively update the members dependent on it using Equation (35).

\[ \text{in}(i, p, nV , oV) = \{(\text{newIRI}_i, \text{return}(p, \text{propertyMappings}), \text{replace}(\text{targetValue}(i, p), oV, nV)) \} \]

\[ \text{update2Insert}(\text{iri}_i, \text{iri}_n, \text{cp} ) = \{(\text{iri}_o, \text{type2:status}, “Expired”), (\text{iri}_o, \text{type2:toDate}, \text{sysdate}()-1), (\text{iri}_o, \text{type2:fromDate}, \text{sysdate}()), (\text{iri}_o, \text{type2:toDate}, “9999 - 12 - 31”), (\text{iri}_o, \text{type2:status}, “Current”) \} \]

\[ \text{retrieveTriples}(\text{iri}_o, \text{cp}) = \text{EXECUTE QUERY} ((\text{iri}_o, ?p, ?v) \text{FILTER}(\text{?p} \notin (\text{cp}, \text{type2:toDate}, \text{type2:fromDate}, \text{type2:status})), \text{tABox}, (\text{iri}_o, ?p, ?v)) \]

\[ \text{recursiveUpdate}(i, p) = \bigcup_{(i, p) \in \text{getAssociates}(\text{iri}_o)} \text{recursiveUpdate}(i, p) \]

\[ \text{update2Insert}(i, \text{iri}_i, \text{iri}_n, \text{cp}) = \{ \text{(iri}_o, \text{type2:status}, “Expired”), \}

\[ \text{(iri}_o, \text{type2:toDate}, \text{sysdate}()-1), \]

\[ \text{(iri}_o, \text{type2:fromDate}, \text{sysdate}()), \]

\[ \text{(iri}_o, \text{type2:toDate}, “9999 - 12 - 31”), \]

\[ \text{(iri}_o, \text{type2:status}, “Current”) \} \]

\[ \text{getAssociates}(\text{iri}_o) = \text{EXECUTE QUERY} ((\text{iri}_o, ?p, ?v) \text{AND} (?i, \text{rdf:type}, \text{qb4o:LevelMember})), \text{tABox}, (?i, ?p) \}

\[ \text{recursiveUpdate}(i, p) = \{ \text{(updateIRI}(i, \text{iri}_i), \text{iri}_i, \text{iri}_n) \} \]

\[ \text{update2Insert}(i, \text{iri}_i, \text{iri}_n, \text{cp}) \]

This operation updates the target ABox, \( \text{tABox} \).

We refer to [44] for the semantics of the \textit{ExternalLinking} operation and to [26] for the semantics of the \textit{MaterializeInference} operation.

References


R.P. Deb Nath et al. / High-level ETL for semantic data warehouses 131


R.P. Deb Nath et al. / High-level ETL for semantic data warehouses


