Semantic Web 13 (2022) 85–132
DOI 10.3233/SW-210429
IOS Press

High-level ETL for semantic data warehouses

Rudra Pratap Deb Nath [a,b,c,*], Oscar Romero [b], Torben Bach Pedersen [a] and Katja Hose [a]
[a] *Department of Computer Science, Aalborg University, Denmark*
E-mails: rudra@cs.aau.dk, tbp@cs.aau.dk, khose@cs.aau.dk
[b] *Department of Service and Information System Engineering, Universitat Politècnica de Catalunya, Spain*
E-mails: rudra@essi.upc.edu, oromero@essi.upc.edu
[c] *Department of Computer Science and Engineering, University of Chittagong, Bangladesh*
E-mail: rudra@cu.ac.bd

Editor: Philippe Cudre-Mauroux, University of Fribourg, Switzerland
Solicited reviews: Patrick Schneider, Siemens, Austria; five anonymous reviewers

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 ($SETL_{CONSTRUCT}$), based on the innovative ETL constructs proposed here. To evaluate $SETL_{CONSTRUCT}$, 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 $SETL_{PROG}$ in terms of productivity, development time, and performance. The evaluation shows that 1) $SETL_{CONSTRUCT}$ uses 92% fewer Number of Typed Characters (NOTC) than $SETL_{PROG}$, and $SETL_{AUTO}$ (the extension of $SETL_{CONSTRUCT}$ for generating ETL execution flows automatically) further reduces the Number of Used Concepts (NOUC) by another 25%; 2) using $SETL_{CONSTRUCT}$, the development time is almost cut in half compared to $SETL_{PROG}$, and is cut by another 27% using $SETL_{AUTO}$; and 3) $SETL_{CONSTRUCT}$ is scalable and has similar performance compared to $SETL_{PROG}$. We also evaluate our approach qualitatively by interviewing two ETL experts.

Keywords: Semantic ETL, Semantic Data Warehosue, 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,[1] Linked Open Data Cloud[2] (LOD), etc. Most SW data provided by international and governmental organizations include facts and fig-

[*]Corresponding author. E-mail: rudra@cs.aau.dk.

[1]https://datahub.io/
[2]https://lod-cloud.net/

1570-0844 © 2022 – The authors. Published by IOS Press. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0).

ures, which give rise to new requirements for Business Intelligence tools to enable analyses in the style of Online 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 Data[3] 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 $SETL_{PROG}$), a programmable semantic ETL framework that semantically integrates both semantic and non-semantic data sources. In $SETL_{PROG}$, 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-

[3]https://opengovdata.org/

ferent to $SETL_{PROG}$ 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[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 $SETL_{CONSTRUCT}$, based on the innovative ETL constructs proposed here. $SETL_{CONSTRUCT}$ allows creating ETL flows by dragging, dropping, and connecting the ETL operations. In addition, it allows creating ETL data flows automatically (we call it $SETL_{AUTO}$).
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 $SETL_{CONSTRUCT}$ improves considerably over $SETL_{PROG}$ in terms of productivity, development time, and performance. In summary: 1) $SETL_{CONSTRUCT}$ uses 92% fewer Number of Typed Characters (NOTC) than $SETL_{PROG}$, and $SETL_{AUTO}$ further reduces the Number of Used Concepts (NOUC) by another 25%; 2) using $SETL_{CONSTRUCT}$, the development time is almost cut in half compared to $SETL_{PROG}$, and is cut by another 27% using $SETL_{AUTO}$; 3) $SETL_{CONSTRUCT}$ is scalable and has similar performance compared to $SETL_{PROG}$. Additionally, we interviewed two ETL experts to evaluate our approach qualitatively.

[4]Here, we use the terms "target" and "MD SDW" interchangeably.

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 $SETL_{CONSTRUCT}$ and compare the process with $SETL_{PROG}$ 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 $\xrightarrow{\text{predicate}}$ 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 vocabulary of a domain, and the ABox is the assertions of the TBox. The TBox is formally defined as a 3-tuple: TBox $= (C, P, A^O)$, where C, P, and A^O are the sets of concepts, properties, and terminological 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 $C(T) = \{c|type(c) \in \mathbb{P}(\{\texttt{rdfs:Class}, \texttt{owl:Class}\})\}$ and $P(T) = \{p|type(p) \in \mathbb{P}(\{\texttt{rdf:Property}, \texttt{owl:ObjectProperty}, \texttt{owl:DatatypeProperty}\})\}$, respectively, where $type(x)$ returns the set of concepts of x, i.e., $(x$ `rdf:type` $?type(x))$ – 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-`

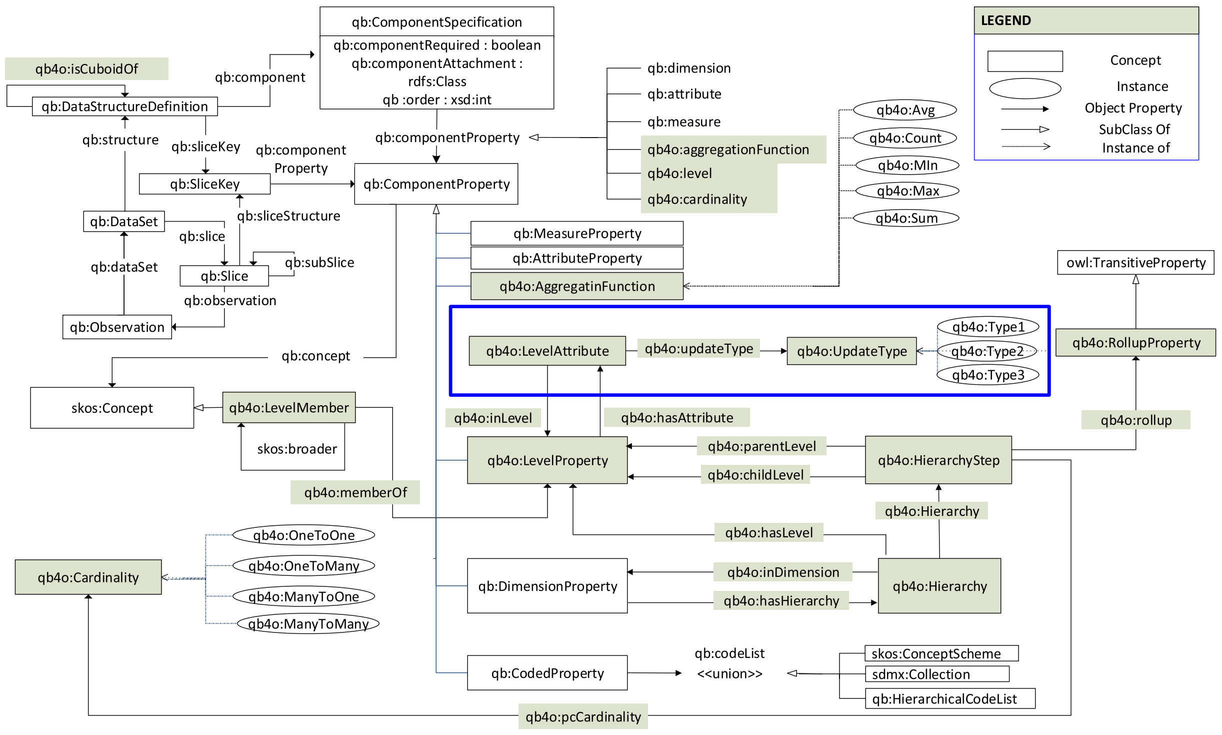


Fig. 1. QB4OLAP vocabulary.

perty, `qb4o:LevelProperty`, and `qb4o:Hierarchy` are used. A dimension can have one or more hierarchies. The relationship between a dimension and its hierarchies are 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 are 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

$$\begin{aligned} C(T) &\\ &= \{c \mid type(c) \in \mathbb{P}(\{\texttt{rdfs:Class}, \texttt{owl:Class}, \\ &\quad \texttt{qb:DataStructureDefinition}, \\ &\quad \texttt{qb:DataSet}, \texttt{qb:DimensionProperty}, \\ &\quad \texttt{qb4o:LevelProperty}, \\ &\quad \texttt{qb4o:Hierarchy}, \\ &\quad \texttt{qb4o:HierarchyStep}\})\} \qquad (1) \end{aligned}$$

$$P(T) = \{p \mid type(p) \in \mathbb{P}(\{\texttt{rdf:Property}, \texttt{owl:ObjectProperty}, \texttt{owl:DatatypeProperty}, \texttt{qb4o:LevelAttribute}, \texttt{qb:MeassureProperty}, \texttt{qb4o:Rollup Property}\})\} \quad (2)$$

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/. 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

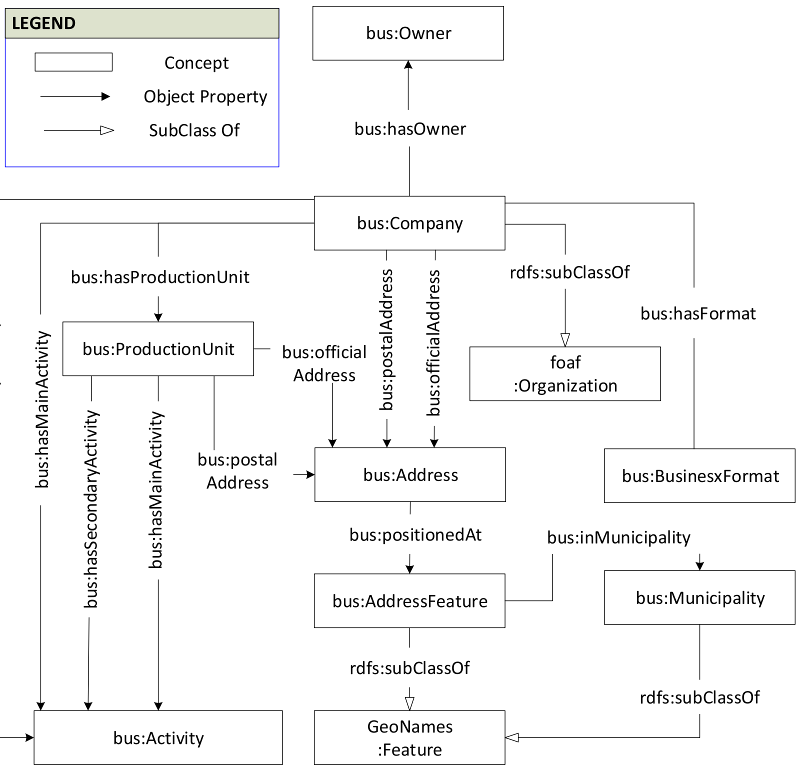


Fig. 2. The ontology of the Danish Business dataset. Due to the large number of datatype properties, they are not included.

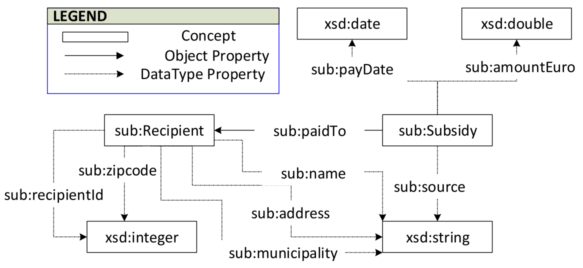


Fig. 3. The ontology of the Subsidy dataset. Due to the large number of datatype properties, all are not included.

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 *SETL$_{PROG}$* 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.

```
### Business Dataset
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bus: <http://extbi.lab.aau.dk/ontology/business/>
PREFIX company: <http://extbi.lab.aau.dk/ontology/
                                        business/Company#>
PREFIX activity: <http://extbi.lab.aau.dk/ontology/
                                        business/Activity#>
PREFIX businessType: <http://extbi.lab.aau.dk/ontology/
                                   business/BusinessType#>
PREFIX owner: <http://extbi.lab.aau.dk/ontology/
                                        business/Owner#>
## 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:4000175029_10058996;
   bus:ownerName "Kim Jonni Larsen";
   bus:officialaddress "Valsomaglevej 117, Ringsted".
company:10165164 rdf:type bus:Company;
   bus:companyId 10165164;
   bus:name "Idomlund 1 Vindmollelaug I/S";
   bus:mainActivity activity:351100;
   bus:hasFormat businessType:Interessentskab;
   bus:hasOwner  owner:4000170495_10165164;
   bus:ownerName "Anders Kristian Kristensen";
   bus:officialaddress "Donskaervej 31,Vemb".
---------------------------------------------------------
---------------------------------------------------------
### Subsidy Dataset
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sub: <http://extbi.lab.aau.dk/ontology/subsidy/>
PREFIX recipient: <http://extbi.lab.aau.dk/ontology/
                                   subsidy/Recipient#>
PREFIX subsidy: <http://extbi.lab.aau.dk/ontology/
                                   subsidy/Subsidy#>
## Example of sub:Recipient instances.
recipient:291894 rdf:type sub:Recipient;
   sub:name "Kristian Kristensen";
   sub:address "Donskaervej 31,Vemb";
   sub:municipality "Holstebro";
   sub:recipientID 291894;
   sub:zipcode 7570.
## Example of sub:Subsidy instances.
subsidy:10615413 rdf:type sub:Subsidy;
   sub:paidTo recipient:291894;
   sub:amountEuro "8928.31";
   sub:payDate "2010-05-25".
```

Listing 1. Example instances of the DBD and the Subsidy dataset

4. 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 multidimensional constructs from the integrated TBox and annotate them with the QB4OLAP vocabulary. In the

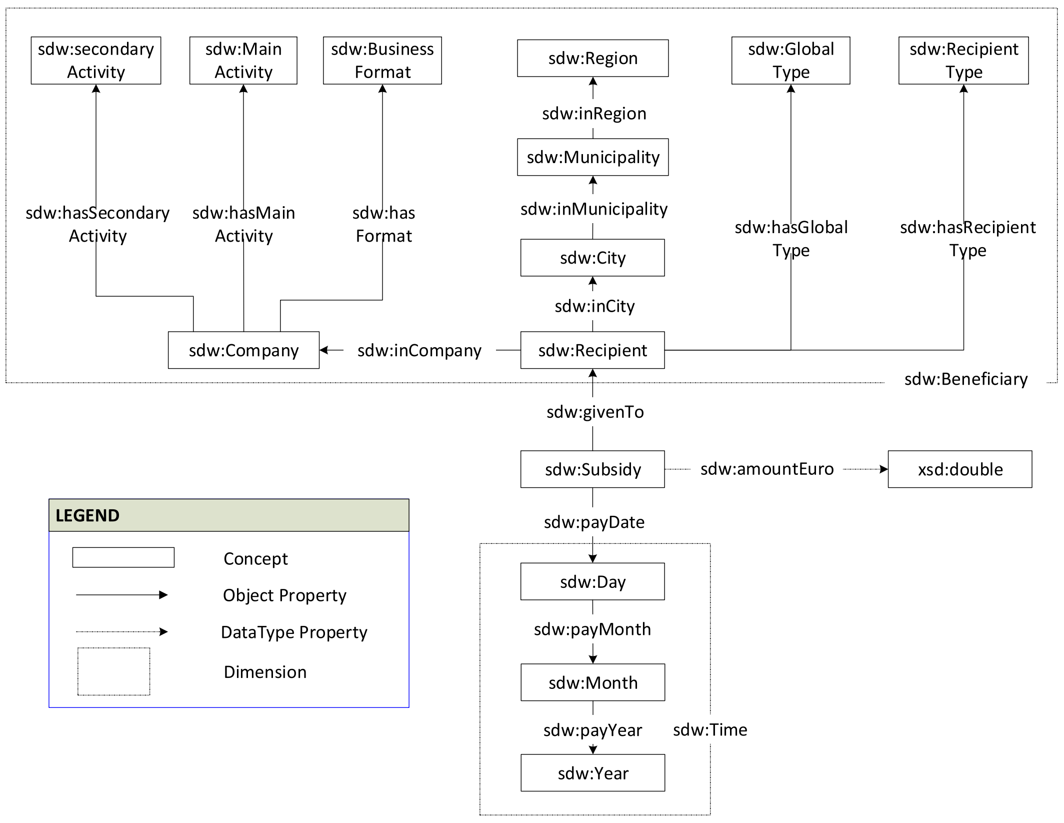


Fig. 4. The ontology of the MD SDW. Due to the large number, data properties of the dimensions are not shown.

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.

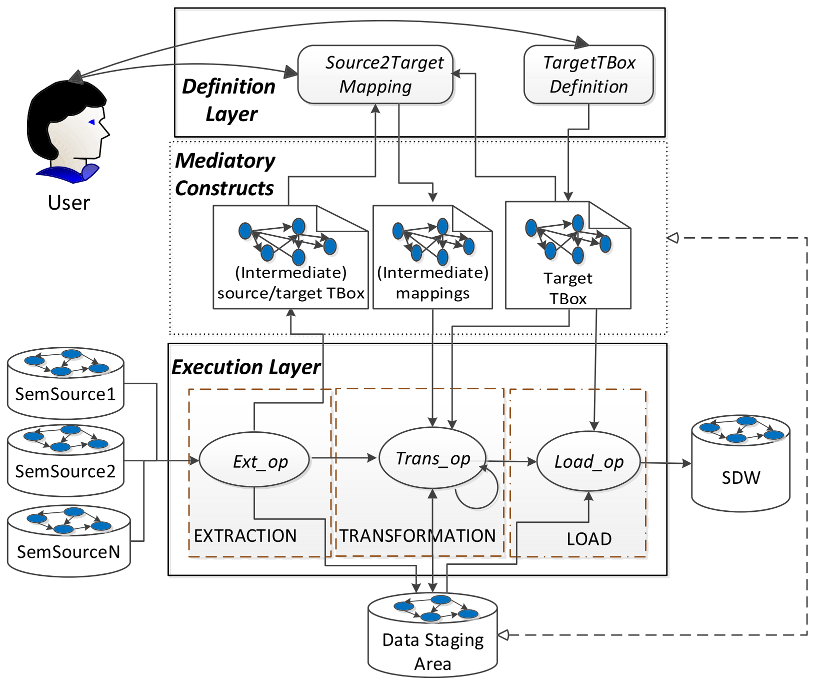


Fig. 5. The overall semantic data integration process. Here, the round-corner rectangle, data stores, dotted boxes, ellipses, and arrows indicate the tasks, semantic data sources and SDW, the phases of the ETL process, ETL operations and flow directions.

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 Mediatory 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, Ext_{op}, $Trans_{op}$, and $Load_{op}$ 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_1's out-

Table 1
Summary of the ETL operations

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

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 $(D_{name}, \mathcal{L}, \rightarrow, \mathcal{H}, \mathcal{F}_{\mathcal{R}})$ 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 $(L_{All}, \emptyset) \in \mathcal{L}$;

- $\rightarrow$ 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 drills 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_{name}, H_L) where H_{name} is the name of a hierarchy and $H_L \subseteq \mathcal{L}$ is the set of levels composing the hierarchy. The directed subgraph formed by this set of levels must be connected; and
- $RUP_{L_i}^{L_j}$ is the property used to relate instances based on the roll-up relationship between L_i and L_j within an hierarchy. $\mathcal{F}_{\mathcal{R}}$ 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_{name} = `sdw:Time`;
2. $\mathcal{L}$ = {(`sdw:Day`, ⟨`sdw:dayId`, `sdw:dayName`⟩), (`sdw:Month`, ⟨`sdw:monthId`, `sdw:monthName`⟩), (`sdw:Year`, ⟨`sdw:yearId`, `sdw:yearName`⟩)};
3. $(\rightarrow)$ = {(`sdw:Day`, `sdw:Month`), (`sdw:Month`, `sdw:Year`), (`sdw:Year`, `sdw:All`)},
4. $\mathcal{H}$ = {`sdw:TimeHierarchy`, {`sdw:Day`, `sdw:Month`, `sdw:Year`, `sdw:All`}}; and
5. $\mathcal{F}_{\mathcal{R}}$ = {
$RUP_{\texttt{sdw:Day}}^{\texttt{sdw:Month}}$ = `sdw:payMonth`,
$RUP_{\texttt{sdw:Month}}^{\texttt{sdw:Year}}$ = `sdw:payYear`,
$RUP_{\texttt{sdw:Year}}^{\texttt{sdw:All}}$ = `sdw:payAll`}.

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

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

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

1. C_{name} = `sdw:Subsidy`;
2. D_L = {`sdw:Day`, `sdw:Recipient`};
3. $\mathcal{M}$ = {`sdw:amounteuro`}; and
4. $\mathcal{F}_{\mathcal{A}}$ = {(`sdw:amounteuro`, SUM), (`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, $ABox$ is defined as $f_{ABox2TBox}(ABox) \rightarrow TBox$. 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 $ABox$ where `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 `rdfs:subClassOf` C_1); 2) if they do not share any instances, they are disjoint (C_1 `owl:disjointWith` C_2); and 3) if C_1 and C_2

```
PREFIX sdw: <http://extbi.lab.aau.dk/ontology/sdw/>
PREFIX rdf:     http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX rdfs:    http://www.w3.org/2000/01/rdf-schema#
PREFIX qb: <http://purl.org/linked-data/cube#>
PREFIX qb4o: <http://purl.org/qb4olap/cubes#>

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

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

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

# Level attributes
sdw:dayId rdf:type qb4o:LevelAttribute;
       rdfs:label "day ID";
       qb4o:updateType qb4o:Type2;
       rdfs:range xsd:String.
sdw:monthId rdf:type qb4o:LevelAttribute;
       rdfs:label "Month ID";
       qb4o:updateType qb4o:Type2;
       rdfs:range xsd:String.
sdw:yearId rdf:type qb4o:LevelAttribute;
       rdfs:label "year ID";
       qb4o:updateType qb4o:Type2;
       rdfs:range xsd:String.
sdw:dayName rdf:type qb4o:LevelAttribute;
       rdfs:label "day Name";
       qb4o:updateType qb4o:Type1;
       rdfs:range xsd:String.
sdw:monthName rdf:type qb4o:LevelAttribute;
       rdfs:label "Month Name";
       qb4o:updateType qb4o:Type1;
       rdfs:range xsd:String.
sdw:yearName rdf:type qb4o:LevelAttribute;
       rdfs:label "year Name";
       qb4o:updateType qb4o:Type1;
       rdfs:range xsd:String.

#roll-up relations
sdw:payMonth rdf:type qb4o:RollupProperty.
sdw:payYear rdf:type qb4o:RollupProperty.
sdw:payAll rdf:type qb4o:RollupProperty.

# Hierarchy Steps
_:ht1 rdf:type qb4o:HierarchyStep;
      qb4o:inHierarchy sdw:TimeHierarchy;
      qb4o:childLevel sdw:Day;
      qb4o:parentLevel sdw:Month;
      qb4o:pcCardinality qb4o:OneToMany;
      qb4o:rollup sdw:payMonth.
_:ht2 rdf:type qb4o:HierarchyStep;
      qb4o:inHierarchy sdw:TimeHierarchy;
      qb4o:childLevel sdw:Month;
      qb4o:parentLevel sdw:Year;
      qb4o:pcCardinality qb4o:OneToMany;
      qb4o:rollup sdw:payYear.
_:ht2 rdf:type qb4o:HierarchyStep;
      qb4o:inHierarchy sdw:TimeHierarchy;
      qb4o:childLevel sdw:Year;
      qb4o:parentLevel sdw:All;
      qb4o:pcCardinality qb4o:OneToMany;
      qb4o:rollup sdw:payAll.

## Subsidy Cube
sdw:amounteuro rdf:type qb:MeasureProperty;
      rdfs:label "subsidy amount"; rdfs:range xsd:Double.
sdw:SubsidyStructure rdf:type qb:DataStructureDefinition;
      qb:component[qb4o:level sdw:Recipient];
      qb:component[qb4o:level sdw:Day];
      qb:component[qb:measure sdw:amounteuro;
           qb4o:aggregateFunction qb4o:sum, qb4o:avg].
# Subsidy Dataset
sdw:SubsidyMD rdf:type qb:Dataset;
      rdfs:label "Subsidy dataset";
      qb:structure sdw:SubsidyStructure;
```

Listing 2. QB4OLAP representation of `sdw:Time` dimension and `sdw:Subsidy` cube

are both a subclass of each other, then they are equivalent (C_1 `owl:equivalentClass` C_2).

3. P, D, R: By checking the unique predicates of the triples, P is derived. A property $p \in P$ 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 datatype 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 formally defines a source-to-target mapping.

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):

$$\begin{aligned}&\mathit{SourceToTargetMapping}(T_S, T_T)\\&\quad = \{(c_s, \mathit{relation}, c_t, \mathit{loc}_{c_s}, \mathit{loc}_{c_t}, \mathit{mapIns}, p_{map},\\&\qquad \mathit{tin}_{iri}, p_{com}, \mathit{op})\}.\end{aligned}$$

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

- $c_s \in \mathcal{C}(T_S)$ and $c_t \in \mathcal{C}(T_T)$ are a source and a target concept respectively, where $\mathcal{C}(T)$ defined in Equation (1).
- *relation* $\in \{\equiv, \sqsubseteq, \sqsupseteq, \bowtie, ⟖, ⟕\}$ represents the relationship between the source and target concept. The relationship can be either *equivalence* ($c_s \equiv c_t$), *supersumption* ($c_s \sqsupseteq c_t$), *subsumption* ($c_s \sqsubseteq c_t$), or *join*. A join relationship can be either a natural join ($c_s \bowtie c_t$), a right-outer join ($c_s ⟖ c_t$), or a left-outer join ($c_s ⟕ c_t$). 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.
- loc_{c_s} and loc_{c_t} are the locations of source and target concept ABoxes.
- $\mathit{mapIns} \in (\{\mathit{All}\} \cup \mathit{FilterCondition})$ 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.
- $p_{map} = \{(p_{c_s}, p_{c_t})\}$ is a set of property-mappings across the properties of c_s and c_t. p_{c_s} can be a property from $\mathit{property}(c_s)$ or an expression over the elements of $\mathit{exp}(\mathit{property}(c_s) \cup \mathit{property}(c_t))$ and p_{c_t} is a property from $\mathit{property}(c_t)$. Here, $\mathit{property}(c)$ returns the union of the set of properties which are connected with concept c either using the `rdfs:domain` or `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.
- tin_{iri} 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 $\mathit{exp}(\mathit{property}(c_s))$, or in an incremental way.
- $p_{com} = \{(\mathit{scom}_i, \mathit{tcom}_i) | \mathit{scom}_i \in \mathit{property}(e_{s_i}), \mathit{tcom}_i \in \mathit{property}(e_{t_i})\}$ 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_{com} is required when the relationship between the source and target concept is a join.
- *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 *op* is a sequence of ETL operations, the location of the input ABox location for the first operation in the sequence is 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.

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

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

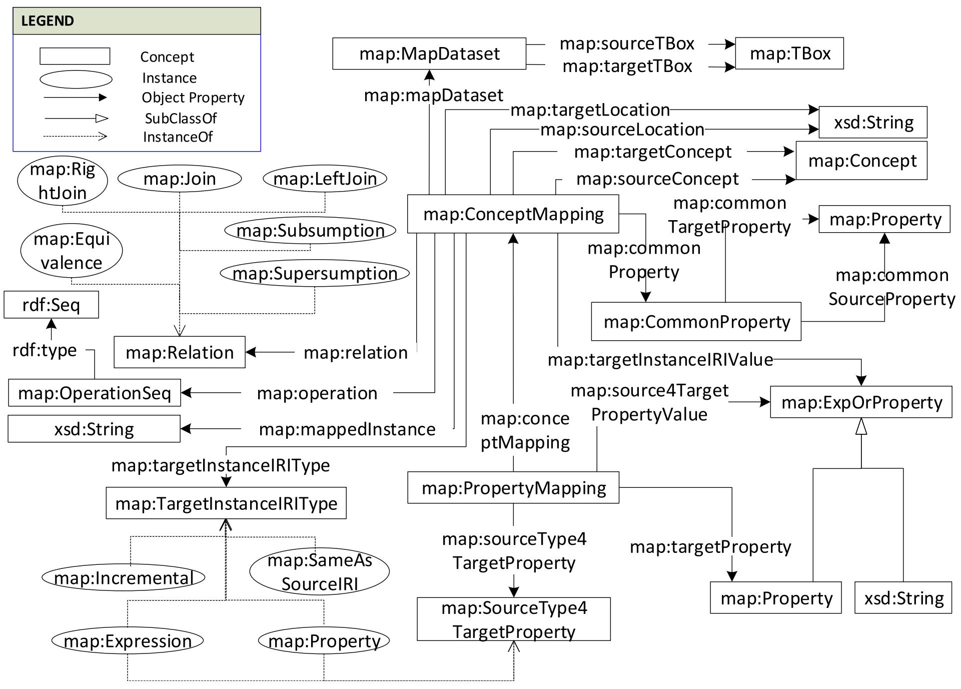


Fig. 6. Graphical overview of key terms and their relationships to the S2TMAP vocabulary.

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` property, 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.

```
PREFIX onto: <http://extbi.lab.aau.dk/ontology/>
PREFIX bus:  <http://extbi.lab.aau.dk/ontology/business/>
PREFIX sub:  <http://extbi.lab.aau.dk/ontology/subsidy/>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX map:  <http://extbi.lab.aau.dk/ontology/s2tmap/>
PREFIX sdw:  <http://extbi.lab.aau.dk/sdw>
PREFIX : <http://extbi.lab.aau.dk/ontology/s2map/example#>
## MapDataset
:mapDataset1 rdf:type map:Dataset;
  rdfs:label "Map-dataset for business and subsidy ontology";
  map:sourceTBox "/map/businessTBox.ttl";
  map:targetTBox "/map/subsidyTBox.ttl".
:mapDataset2 rdf:type map:Dataset;
  rdfs:label "Map-dataset for subsidy and subsidyMD ontology";
  map:sourceTBox "/map/subsidyTBox.ttl";
  map:targetTBox "/map/subsidyMDTBox.ttl".
##ConceptMapping: Joining  Recipient and Company
:Recipient_Company rdf:type map:ConceptMapping;
  rdfs:label "join-transformation between
          bus:Company and sub:Recipient";
  map:mapDataset :mapDataset1;
  map:sourceConcept bus:Company;
  map:targetConcept sub:Recipient;
  map:sourceLocation "/map/dbd.nt";
  map:targetLocation "/map/subsidy.nt";
  map:relation map:RightJoin;
  map:mappedInstance "All";
  map:targetInstanceIRIUniqueValueType map:SameAsSourceIRI;
  map:operation _:opSeq;
  map:commonProperty _:cp1, _:cp2.
_:opSeq  rdf:type rdf:Seq;
  rdf:_1 map:joinTransformation.
_:cp1 map:sourceCommonProperty bus:ownerName;
  map:targetCommonProperty sub:name.
_:cp2 map:sourceCommonProperty bus:officialAddress;
  map:targetCommonProperty sub:address.

#concept-mapping: Populating the sdw:Recipient level
:Recipient_RecipientMD   rdf:type map:ConceptMapping;
  rdfs:label "Level member generation";
  map:mapDataset :mapDataset2;
  map:sourceConcept sub:Recipient;
  map:targetConcept sdw:Recipient;
  map:sourceLocation "/map/subsidy.nt";
  map:targetLocation "/map/sdw";
  map:relation map:Equivalence;
  map:mappedInstance "All";
  map:targetInstanceIRIValueType map:Property;
  map:targetInstanceIRIValue sub:recipientID;
  map:operation _:opSeq1.
_:opSeq1 rdf:type rdf:Seq;
  rdf:_1 map:LevelMemberGenerator;
  rdf:_2 map:Loader.
#concept-mapping: Populating the cube dataset
:Subsidy_SubsidyMD rdf:type map:ConceptMapping;
  rdfs:label "Observation generation";
  map:mapDataset :mapDataset2;
  map:sourceConcept sub:Subsidy;
  map:targetConcept sdw:SubsidyMD;
  map:sourceLocation "/map/subsidy.nt";
  map:targetLocation "/map/sdw";
  map:relation owl:equivalentClass;
  map:mappedInstance "All";
  map:targetInstanceIRIUniqueValueType map:Incremental;
  map:operation _:opSeq2.
_:opSeq2 rdf:type rdf:Seq;
  rdf:_1 map:GraphExtractor;
  rdf:_2 map:TransformationOnLiteral;
  rdf:_3 map:ObservationGenerator;
  rdf:_4 map:Loader.
## property-mapping under :Recipient_Company
:companyID_companyID rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for companyID";
  map:conceptMapping :Recipient_Company;
  map:targetProperty sub:companyId;
  map:sourceType4TargetProperty map:Property;
  map:source4TargetPropertyValue bus:companyId.
:businessType_businessType rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for business type";
  map:conceptMapping :Recipient_Company;
  map:targetProperty sub:businessType;
  map:sourceType4TargetProperty map:Property;
  map:source4TargetPropertyValue bus:hasFormat.
:address_city rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for city";
  map:conceptMapping :Recipient_Company;
  map:targetProperty sub:cityId;
  map:sourceType4TargetProperty map:Expression;
  map:source4TargetPropertyValue STRAFTER(sub:address,",").
:name_name rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for name";
  map:conceptMapping :Recipient_Company;
  map:targetProperty sub:name;
  map:sourceType4TargetProperty map:Property;
  map:source4TargetPropertyValue sub:name.
# property-mappings under :Recipient_RecipientMD
:companyId_company rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for companyId";
  map:conceptMapping :Recipient_RecipientMD;
  map:targetProperty sdw:hasCompany;
  map:sourceType4TargetProperty map:Property;
  map:source4TargetPropertyValue sub:companyId;
:cityId_city rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for cityId";
  map:conceptMapping :Recipient_RecipientMD;
  map:targetProperty sdw:inCity;
  map:sourceType4TargetProperty map:Property;
  map:source4TargetPropertyValue sub:city;
:name_name rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for name";
  map:conceptMapping :Recipient_RecipientMD;
  map:targetProperty sdw:name;
  map:sourceType4TargetProperty map:Property;
  map:source4TargetPropertyValue sub:name
# property-mappings under :Subsidy_SubsidyMD
:Recipient_recipientId rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for recipient in sdw:Subsidy";
  map:conceptMapping :Subsidy_SubsidyMD;
  map:targetProperty sdw:Recipient;
  map:sourceType4TargetProperty map:Property;
  map:source4TargetPropertyValue sub:paidTo.
:hasPayDate_Day rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for Day of sdw:SubsidyMD";
  map:conceptMapping :Subsidy_SubsidyMD;
  map:targetProperty sdw:Day;
  map:sourceType4TargetProperty map:Expression;
  map:source4TargetPropertyValue
  "CONCAT(STR(DAY(sub:payDate)),"/",
   STR(MONTH(sub:payDate)),"/",STR(YEAR(sub:payDate)))".
:amountEuro_amountEuro rdf:type map:PropertyMapping;
  rdfs:label "property-mapping for amountEuro measure";
  map:conceptMapping :Subsidy_SubsidyMD;
  map:targetProperty sdw:amountEuro;
  map:sourceType4TargetProperty map:Property;
  map:source4TargetPropertyValue sub:amountEuro.
```

Listing 3. An S2TMAP representation of the mapping file of our use case

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 shows 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,

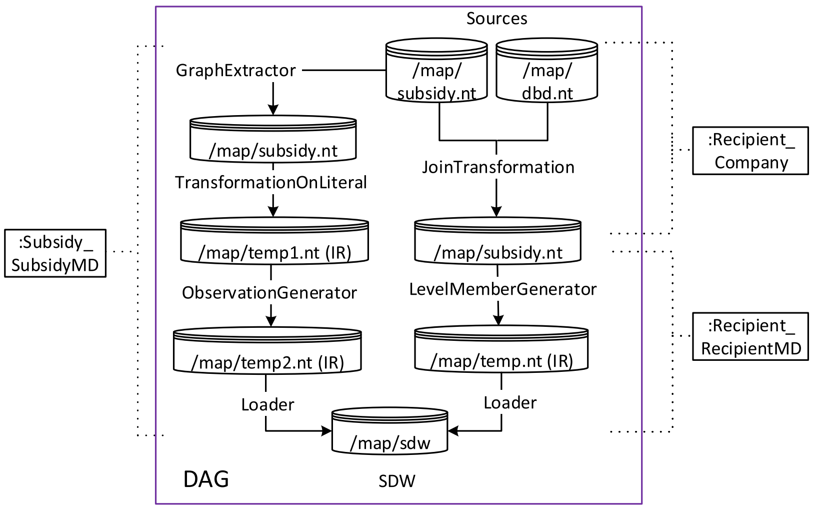


Fig. 7. The conceptual presentation of Listing 3.

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[5] instead of returning output.

Developers can use either of the two following options: (i) The recommended option is that given a TBox construct *aConstruct* (a concept, a level, or a QB dataset) and a mapping file *aMappings* generated in the Definition Layer, the automatic ETL execution flow generation process will automatically extract the parameter values from *aMappings* (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 *aMappings* and *aConstruct* 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).

[5]An operation has a side effect if it modifies some state variable value(s) outside its local environment (https://en.wikipedia.org/wiki/Side_effect_(computer_science)).

6.1. Extraction operations

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

GraphExtractor(Q, G, outputPattern, tABox) 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. *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 *aMappings* where *aConstruct* appears (i.e., in *aMapping*, $cm \xrightarrow{map:targetConcept} aConstruct$) and the operation to process *cm* is *GraphExtractor* (i.e., *GraphExtractor* is an element in the operation sequence defined by the `map:operation` property). Then, it parametrizes *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, *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 *GraphExtractor(Q, G, outputPattern, tABox)* operation, where

1. *Q=((?ins,*`rdf:type`*,*`bus:Company`*) AND (?ins,?p,?v))*,[6]
2. *G*="/map/dbd.ttl",
3. *outputPattern= (?ins,?p,?v)*,
4. *tABox*="/map/com.ttl".[7]

Listing 4 shows the output of this operation.

```
company:10058996 rdf:type bus:Company;
    bus:name "Regerupgard v/Kim Jonni Larsen";
    bus:mainActivity activity:11100;
    bus:secondaryActivity activity:682040;
    bus:hasFormat businessType:Enkeltmandsvirksomhed;
    bus:hasOwner  owner:4000175029_10058996;
    bus:ownerName "Kim Jonni Larsen";
    bus:address "Valsomaglevej 117, Ringsted".
company:10165164 rdf:type bus:Company;
    bus:name "Idomlund 1 Vindmollelaug I/S";
    bus:mainActivity activity:351100;
    bus:hasFormat businessType:Interessentskab;
    bus:hasOwner  owner:4000170495_10165164;
    bus:ownerName "Anders Kristian Kristensen";
    bus:address "Donskaervej 31,Vemb".
```

Listing 4. Example of *GraphExtractor*

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

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

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: *TransformationOnLiteral*, *JoinTransformation*, *LevelMemberGenerator*, *ObservationGenerator*, *ChangedDataCapture*, *UpdateLevel*, *External linking*, and *MaterializeInference*. The following describe each operation.

TransformationOnLiteral(sConstruct, tConstruct, sTBox, sABox propertyMappings, tABox) As described in the *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 *aMappings*, where *aConstruct* appears and the operation to process *cm* is *TransformationOnLiteral*. Then, the process parametrizes *TransformationOnLiteral* as follows: 1) *sConstruct* and *tConstruct* are defined by `map:sourceConcept` and `map:targetConcept`; 2) *sTBox* is the target TBox of *cm*'s map-dataset, defined by the property `map:sourceTBox`; 3) *sABox* is the location of the source ABox defined by `map:sourceLocation`; 4) *propertyMappings* is the set of property-mappings defined under *cm*; and 5) *tABox* is the location of the target ABox defined by `map:targetLocation`. A developer can also manually set the parameters. From the given inputs, this operation transforms (or directly returns) the *sABox* triple objects according to the expressions (defined through `map:source4TargetPropertyValue`) in *propertyMappings* and stores the triples in *tABox*. This operation first creates a SPARQL SELECT query based on the expressions defined in *propertyMappings*, and then, on top of the SELECT query, it forms a SPARQL CONSTRUCT query to generate the transformed ABox for *tConstruct*.

```
 1 ## Property-mappings input
 2 :hasPayDate_Day rdf:type map:PropertyMapping;
 3  map:targetProperty sub:hasPayDate;
 4  map:sourceType4TargetPropertyValue map:Expression;
 5  map:source4TargetPropertyValue "CONCAT(STR(DAY(hasPayDate)),
 6  "/", STR(MONTH(hasPayDate)),"/",STR(YEAR(hasPayDate)))".
 7 :Recipient_recipientId rdf:type map:PropertyMapping;
 8  map:targetProperty sub:hasRecipient;
 9  map:sourceType4TargetPropertyValue map:Property;
10  map:source4TargetPropertyValue sub:hasRecipient.
11 :amountEuro_amountEuro rdf:type map:PropertyMapping;
12  map:targetProperty sub:amountEuro;
13  map:sourceType4TargetPropertyValue map:Expression;
14 map:source4TargetPropertyValue "xsd:integer(sub:amountEuro)".
15 ## sub:Subsidy instances after TransformationOnLiteral.
16 subsidy:10615413 rdf:type sub:Subsidy;
17  sub:hasRecipient recipient:291894;
18  sub:amountEuro 8928;
19  sub:hasPayData "25/25/2010".
```

Listing 5. Example of *TransformationOnLiteral*

Example 6. Listing 5 (lines 16–19) shows the transformed instances after applying the operation *TransformationOnLiteral(sConstruct, tConstruct, sTBox, sABox, PropertyMappings, tABox)*, where

1. *sConstruct* = *tConstruct*=`sub:Subsidy`,
2. *sTBox*="/map/subsidyTBox.ttl",
3. *sABox*= source instances of `sub:Subsidy` (lines 47–50 in Listing 1),
4. *propertyMappings* = lines 2–14 in Listing 5,
5. *tABox*="/map/temp1.ttl".

JoinTransformation(sConstruct, tConstruct, sTBox, tTBox, sABox, tABox, comProperty, propertyMappings) A TBox construct (a concept, a level, or a QB dataset) can be populated from multiple sources. Therefore, an operation is necessary to join and transform data coming from different sources. Two constructs of the same or different sources can only be joined if they share some common properties. This operation joins a source and a target constructs based on their common properties and produce the instances of the target construct by resolving the source expressions mapped to target properties. To join *n* sources, an ETL process requires *n-1 JoinTransformation* operations.

If the ETL execution flow is generated automatically, the automatic ETL execution flow generation process first identifies the concept-mapping *cm* from *aMappings*, where *aConstruct* appears and the operation to process *cm* is *JoinTransformation*. Then it parameterizes *JoinTransformation* as follows: 1) *sConstruct* and *tConstruct* are defined by the `map:sourceConcept` and `map:targetConcept` properties; 2) *sTBox* and *tTBox* are the source and target TBox of *cm*'s map-dataset, defined by `map:sourceTBox` and `map:targetTBox`; 3) *sABox* and *tABox* are defined by the `map:sourceLocation` and `map:targetLocation` properties; 4) *comProperty* is defined by `map:commonProperty`; and 5) *propertyMappings* is the set of property-mappings defined under *cm*.

A developer can also manually set the parameters. Once it is parameterized, *JoinTransformation* joins two constructs based on *comProperty*, transforms their data based on the expressions (specified through `map:source4TargetPropertyValue`) defined in *propertyMappings*, and updates *tABox* based on the join result. It creates a SPARQL SELECT query joining two constructs using either `AND` or `OPT` features, and on top of that query, it forms a SPARQL CONSTRUCT query to generate the transformed *tABox*.

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, tTBox, sABox, tABox, comProperty, propertyMappings)*, where

1. *sConstruct*= `bus:Company`,
2. *tConstruct*= `sub:Recipient`,
3. *sTBox*= "/map/businessTBox.ttl",
4. *tTBox*= "/map/subsidyTBox.ttl",
5. *sABox*= source instances of `bus:Company` (lines 13–29 in Listing 1),
6. *tABox*= source instances of `sub:Recipient` (lines 40–45 in Listing 1),
7. *comProperty* = lines 31, 34–37 in Listing 3,
8. *propertyMappings* = lines 73–96 in Listing 3.

Listing 6 shows the output of the *joinTransformation* operation.

```
## Example of sub:Recipient instances.
recipient:291894 rdf:type sub:Recipient;
   sub:name "Kristian Kristensen";
   sub:cityId "Vemb";
   sub:companyId company:10165164;
   sub:businessType businessType:Interessentskab.
```

Listing 6. Example of *JoinTransformation*

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 level[8] 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:sourceTBox` and `map:targetTBox`; 4) *sABox* is the source ABox defined by the property `map:sourceLocation`; 5) *iriValue* is a rule[9] to create IRIs for the level members and it is defined defined by the `map:TargetInstanceIriValue` property; 6) *iriGraph* is the IRI graph[10] 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 paramenters. 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

1. *sConstruct*= `sub:Recipient`,
2. *level*= `sdw:Recipient`,
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. *propertyMappings*= lines 98–115 in Listing 3,
9. *tABox*="/map/temp.ttl".

```
PREFIX sdw: <http://extbi.lab.aau.dk/ontology/sdw/>
PREFIX recipient: <http://extbi.lab.aau.dk/ontology
                                /sdw/Recipient#>
PREFIX company: <http://extbi.lab.aau.dk/ontology
                                  /sdw/Company#>
PREFIX city: <http://extbi.lab.aau.dk/ontology
                                    /sdw/City#>
## Example of a recipient level member.
recipient:291894 rdf:type qb4o:LevelMember;
                 qb4o:memberOf sdw:Recipient.
                 sdw:name "Kristian Kristensen";
                 sdw:inCity city:Vemb;
                 sdw:hasCompany company:10165164.
```

Listing 7. Example of *LevelMemberGenerator*

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",

[8]A level is termed as a level property in QB4OLAP, therefore, throughout this paper, we use both the term "level" and "level property" interchangeably.

[9]A rule can be either a source property, an expression or incremental, as described in Section 5.

[10]The IRI graph is an RDF graph that keeps a triple for each resource in the SDW with their corresponding source IRI.

```
 1 PREFIX sdw: <http://extbi.lab.aau.dk/ontology/sdw/>
 2 PREFIX subsidy: <http://extbi.lab.aau.dk/ontology
 3                                  /sdw/Subsidy#>
 4 PREFIX recipient: <http://extbi.lab.aau.dk/ontology
 5                                   /sdw/Recipient#>
 6 PREFIX day: <http://extbi.lab.aau.dk/ontology/sdw/Day#>
 7 ## Property-Mappings
 8 :recipientId_Recipient rdf:type map:PropertyMapping;
 9   map:targetProperty sdw:Recipient;
10   map:sourceType4TargetPropertyValue map:Property;
11   map:source4TargetPropertyValue sub:hasRecipient.
12 :hasPayDate_Day rdf:type map:PropertyMapping;
13   map:targetProperty sdw:Day;
14   map:sourceType4TargetPropertyValue map:Property;
15   map:source4TargetPropertyValue sub:hasPaydate.
16 :amountEuro_amountEuro rdf:type map:PropertyMapping;
17   map:targetProperty sdw:amountEuro;
18   map:sourceType4TargetPropertyValue map:Property;
19   map:source4TargetPropertyValue sub:amountEuro.
20 ## Example of observations
21 subsidy:_01 rdf:type qb4o:Observation;
22   inDataset sdw:SubsidyMD;
23   sdw:hasRecipient recipient:291894;
24   sdw:amountEuro "8928.00";
25   sdw:hasPayData day:25/25/2010.
```

Listing 8. Example of *ObservationGenerator*

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.

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.

```
 1 ## New snapshot of sub:Recipient instances.
 2 recipient:291894 rdf:type sub:Recipient;
 3                  sub:name "Kristian Jensen";
 4                  sub:cityId "Vemb";
 5                  sub:companyId company:10165164;
 6                  businessType:Interessentskab.
 7 recipient:301894 rdf:type sub:Recipient;
 8                  sub:name "Jack";
 9                  sub:cityId "Aalborg";
10                  sub:companyId company:100000.
11 ## New instances to be inserted
12 recipient:301894 rdf:type sub:Recipient;
13                  sub:name "Jack";
14                  sub:cityId "Aalborg";
15                  sub:companyId company:100000.
16 ## Update triples
17 recipient:291894 sub:name "Kristian Jensen".
```

Listing 9. Example of *ChangedDataCapture*

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).

UpdateLevel(level, updatedTriples, sABox, tTBox, 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 *tTBox* 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 *tTBox* 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, “oldValue”)*.

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) *tTBox* is the target TBox of *cm*’s map-dataset, defined by the property `map:targetTBox`; 5) *tABox* 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).

```
 1 ## sdw:Recipient Level
 2 recipient:762921 rdf:type qb4o:LevelMember;
 3                  qb4o:member sdw:Recipient;
 4                  sdw:name "R. Nielsen";
 5                  sdw:cityId city:Lokken;
 6                  sdw:hasCompany company:10165164.
 7 recipient:291894 rdf:type qb4o:LevelMember;
 8                  qb4o:memberOf sdw:Recipient.
 9                  sdw:name "Kristian Kristensen";
10                  sdw:cityId city:Vemb;
11                  sdw:hasCompany company:10165164.
12 ## After type1 update
13 recipient:762921 rdf:type qb4o:LevelMember;
14                  qb4o:member sdw:Recipient;
15                  sdw:name "R. Nielsen";
16                  sdw:cityId city:Lokken;
17                  sdw:hasCompany company:10165164.
18 recipient:291894 rdf:type qb4o:LevelMember;
19                  qb4o:memberOf sdw:Recipient.
20                  sdw:name "Kristian Jensen";
21                  sdw:cityId city:Vemb;
22                  sdw:hasCompany company:10165164.
23 ##After type2 update
24 recipient:762921 rdf:type qb4o:LevelMember;
25                  qb4o:member sdw:Recipient;
26                  sdw:name "R. Nielsen";
27                  sdw:cityId city:Lokken;
28                  sdw:hasCompany company:10165164.
29
30 recipient:291894 rdf:type qb4o:LevelMember;
31                  qb4o:memberOf sdw:Recipient.
32                  sdw:name "Kristian Kristensen";
33                  sdw:cityId city:Vemb;
34                  sdw:hasCompany company:10165164.
35                  sdw:fromDate ``0000-00-00";
36                  sdw:toDate ``2017-09-25";
37                  sdw:status ``Expired".
38
39 recipient:291894\_2017\_09\_26 rdf:type qb4o:LevelMember;
40                          qb4o:memberOf sdw:Recipient.
41                          sdw:name "Kristian Jensen";
42                          sdw:cityId city:Vemb;
43                          sdw:hasCompany company:10165164;
44                          sdw:fromDate ``2017-09-26";
45                          sdw:toDate ``9999-12-31";
46                          sdw:status ``Current".
47 ## After type3 update
48 recipient:762921 rdf:type qb4o:LevelMember;
49                 qb4o:member sdw:Recipient;
50                 sdw:name "R. Nielsen";
51                 sdw:cityId city:Lokken;
52                 sdw:hasCompany company:10165164.
53
54 recipient:291894 rdf:type qb4o:LevelMember;
55                  qb4o:memberOf sdw:Recipient.
56                  sdw:name "Kristian Jensen";
57                  sdw:name_oldValue "Kristian Kristensen";
58                  sdw:cityId city:Vemb;
59                  sdw:hasCompany company:10165164.
```

Listing 10. Example of different types of updates

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* ∈ *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 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[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 *triplesSet* and the path of a triple store *tsPath*, this operation loads *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 *Loader*. Then, it takes values of `map:sourceLocation` and `map:targetLocation` for the parameters *tripleSet* and *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 Mediatory 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 data flows E. At first, it uses the functions[12] to locate x in G (line 1) and get the concept-mappings where $node_{target}$ 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: 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: CREATEETL

```
Input: TargetConstruct $x$, MappingFile $G$,
       IRIGraph $G_{IRI}$
Output: A set of ETL flows $E$
begin
1   $node_{target} \leftarrow$ FIND($x$, $G$)
2   $?c_{maps} \leftarrow$FINDCONMAP(($node_{target}$, $G$)
    /* In $G$:
       $node_{target} \xleftarrow{\texttt{map:targetConcept}} ?c_{maps}$ ;
       $|?c_{maps}| \geqslant 0$                              */
3   $E \leftarrow$ CREATESET()
4   if ($?c_{maps} \neq \emptyset$) then
5       foreach $c \in ?c_{maps}$ do
6           $s_c \leftarrow$ CREATESTACK()
7           $s_c \leftarrow$ CREATEAFLOW($c$, $s_c$, $G$, $G_{IRI}$)
8           $E$.ADD($s_c$)
9   return $E$
```

Algorithm 2: CREATEAFLOW

```
Input: ConceptMapping $c$, Stack $s_c$, MappingFile
       $G$, IRIGraph $G_{IRI}$
Output: Stack $s_c$
begin
1   $ops \leftarrow$ FINDOPERATIONS($c$, $G$)
    /* In $G$: $c \xrightarrow{\texttt{map:operation}} ops$        */
2   $s_c$. PUSH(PARAMETERIZE($ops$, $c$, $G$, $G_{IRI}$))
    /* Push parameterized
       operations in $s_c$                           */
3   $scon \leftarrow$ FINDSOURCECONCEPT($c$, $G$)
    /* In $G$: $c \xrightarrow{\texttt{map:sourceConcept}} scon$  */
4   $?scon_{map} \leftarrow$ FINDCONMAP($scon$, $G$)
    /* In $G$:
       $scon \xleftarrow{\texttt{map:targetConcept}} ?scon_{map}$ ;
       $|?scon_{map}| \leqslant 1$                          */
5   if ($|?scon_{map}| = 1$) then
6       CREATEAFLOW($nc \in ?scon_{map}$, $s_c$, $G$)
        /* recursive call with $nc$      */
7   $s_c$.PUSH(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 $(i = 1$ 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 metadata 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, $?c_{\text{maps}}$ = {`: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). PARAMETERIZE 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 (*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-*

[13]Here, lines 95–112 in Listing 3.

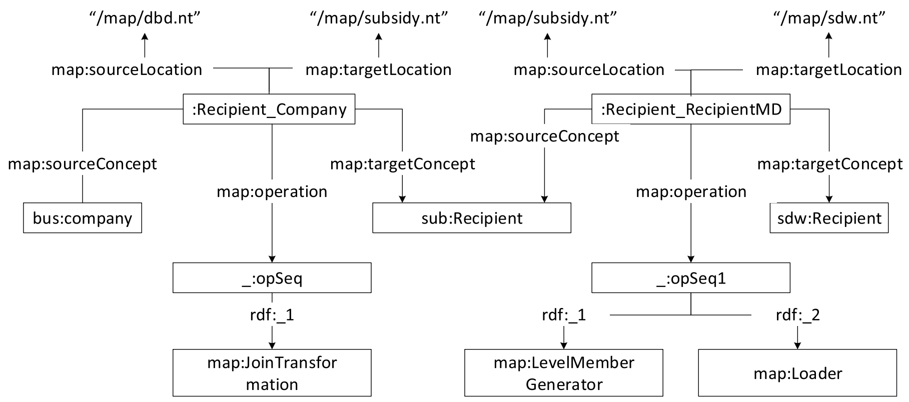


Fig. 8. The graph presentation of a part of Listing 3.

ator(..)[14] is used as the input source location of *Loader(..)*. CREATEAFLOW pushes the parameterized operations to s_c (line 2), hence s_c = (*LevelMemberGenerator(..); Loader(..)*).

Then, CREATEAFLOW finds the source concept of `:Recipient_RecipientMD` (line 3), which is `sub:Recipient`; retrieves the concept-mapping `:Recipient_Company` of `sub:Recipient` from Listing 3 (line 4); and recursively calls itself for `:Recipient_Company` (lines 5–6). The operation sequence of `:Recipient_Company` (line 1) is (*JoinTransformation*) (see Fig. 8), and the call of PARAMETERIZE at line 2 returns the parameterized *JoinTransformation*(`bus:Company`, `Sub:Reicipient`, "/map/dbd.nt", "/map/subsidy.nt", comProperty,[15] propertyMappings[16]) operation, which is pushed to the stack s_c, i.e., s_c = (*JoinTransformation(...); LevelMemberGenerator(...); Loader(...)*). The source concept `bus:Company` is not connected to any other concept-mapping through the `map:targetConcept` property in the mapping file. Therefore, CREATEAFLOW skips lines 5 and 6. After that, it pushes the start operation (StartOp) in s_c, i.e., s_c = (*StartOp, JoinTransformation(..); LevelMemberGenerator(..); Loader(..)*) and returns it to CREATEETL (Algorithm 1) at line 7. Then, CREATEETL adds it to the set of ETL data flows E and returns it (line 9). Therefore, the ETL data flow to populate `sdw:Recipient` is *StartOp* ⇒ *JoinTransformation(..)* ⇒ *LevelMemberGenerator(..)* ⇒ *Loader(..)*.

[14](..) indicates that the operation is parameterized.
[15]lines 32, 36–39 in Listing 3.
[16]Lines 76–93 in Listing 3.

8. Evaluation

We created a GUI-based prototype, named $SETL_{CONSTRUCT}$ [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. $SETL_{CONSTRUCT}$ is developed in Java 8. Like other traditional tools (e.g., PDI [11]), $SETL_{CONSTRUCT}$ allows developers to create ETL flows by dragging, dropping, and connecting the ETL operations. The system is demonstrated in [17]. On top of $SETL_{CONSTRUCT}$, we implement the automatic ETL execution flow generation process discussed in Section 7; we call it $SETL_{AUTO}$. The source code, examples, and developer manual for $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ are available at https://github.com/bi-setl/SETL.

To evaluate $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$, 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 $SETL_{PROG}$[44]. Our evaluation focuses on three aspects: 1) productivity, i.e., to what extent $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ 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 $SETL_{PROG}$ as our competitive system. We could not directly compare $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ 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 supprot 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, $SETL_{PROG}$ supports both semantic and non-semantic source integration, and it uses the relational model as a canonical model. In [44], $SETL_{PROG}$ 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

$SETL_{PROG}$ requires Python knowledge to maintain and implement an ETL process. On the other hand, using $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$, 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 $SETL_{PROG}$, $SETL_{CONSTRUCT}$, and $SETL_{AUTO}$. We measure the developer effort in terms of Number of Typed Characters (NOTC) for $SETL_{PROG}$ and in Number of Used Concepts (NOUC) for $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$. Here, we define a concept as a GUI component of $SETL_{CONSTRUCT}$ 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, required procedures/data structures, number of the task used in the ETL process (NUEP), and NOTC for $SETL_{PROG}$; 2) the required task/operation, NOUC, and number of clicks, selections, and NOTC (for each concept) for $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$. 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 $SETL_{PROG}$, 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 $SETL_{PROG}$, we used 24,509 NOTC. On the other hand, $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ use the *TargetTBoxDefinition* interface to create/edit a target TBox. To create the target TBox of our SDW in $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$, we use 101 concepts that require only 382 clicks, 342 selections, and 1905 NOTC. Therefore, for creating a target TBox, $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ use 92% fewer NOTC than $SETL_{PROG}$.

To create source-to-target mappings, $SETL_{PROG}$ 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, $SETL_{PROG}$ uses 2052 NOTC for creating mappings for the whole ETL process. On the contrary, $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ use a GUI. In total, $SETL_{CONSTRUCT}$ uses 87 concepts composed of 330 clicks, 358 selections, and 30 NOTC. Since $SETL_{AUTO}$ internally creates the intermediate mappings, there is no need to create separate mapping datasets and concept-mappings for intermediate results. Thus, $SETL_{AUTO}$ uses only 65 concepts requiring 253 clicks, 274 selections, and 473 NOTC. Therefore, $SETL_{CONSTRUCT}$ reduces NOTC of $SETL_{PROG}$ by 98%. Although $SETL_{AUTO}$ uses 22 less concepts than $SETL_{CONSTRUCT}$, $SETL_{CONSTRUCT}$ reduces NOTC of $SETL_{AUTO}$ by 93%. This is because, in $SETL_{AUTO}$, we write the data extraction queries in the concept-mappings where in $SETL_{CONSTRUCT}$ we set the data extraction queries in the ETL operation level.

To extract data from either an RDF local file or an SPARQL endpoint, $SETL_{PROG}$ uses the *query()* procedure and the *ExtractTriplesFromEndpoint()* class. On the other hand, $SETL_{CONSTRUCT}$ 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. $SETL_{PROG}$ uses different functions/procedures from the Petl Python library (integrated with $SETL_{PROG}$) based on the cleansing requirements. In $SETL_{CONSTRUCT}$, all data cleansing related tasks on data sources are done using *TransformationOnLiteral* (single source) and *JoinTransformation* (for multi-source). *TransformationOnLiteral*

Table 2

Comparison among the productivity of $SETL_{PROG}$, $SETL_{CONSTRUCT}$, and $SETL_{AUTO}$ for the SDW

Tool ⇒		$SETL_{PROG}$				$SETL_{CONSTRUCT}$				$SETL_{AUTO}$			
						NOUC	Components of Each Concept			NOUC	Components of Each Concept		
ETL Task ⇓	Sub construct	Procedures/Data structures	NUEP	NOTC	Task/Operation		Clicks	Selections	NOTC		Clicks	Selections	NOTC
TBox with MD Semantics	Cube Structure	*Concept()*, *Property()*, *BlankNode()*, *Namespaces()*, *conceptPropertyBindings()*, *createOntology()*	1	665	TargetTBox-Definition	1	9	9	7	1	9	9	7
	Cube Dataset		1	67		1	2	2	18	1	2	2	18
	Dimension		2	245		2	3	4	12	2	3	4	12
	Hierarchy		7	303		7	4	6	18	7	4	6	18
	Level		16	219		16	4	5	17	16	4	5	17
	Level Attribute		38	228		38	4	3	15	38	4	3	15
	Rollup Property		12	83		12	2	1	7	12	2	1	7
	Measure Property		1	82		1	4	3	25	1	4	3	25
	Hierarchy Step		12	315		12	6	6	6	12	6	6	6
	Prefix		21	74		21	1	0	48	21	1	0	48
Subtotal (for the Target TBox)			**1**	**24509**		**101**	**382**	**342**	**1905**	**101**	**382**	**342**	**1905**
Mapping Generation	Mapping Dataset	*N/A*	0	0	SourceTo-TargetMapping	5	1	2	6	3	1	2	6
	Concept-mapping	*dict()*	17	243		23	9	10	0	18	12	10	15
	Property-mapping	*N/A*	0	0		59	2	2	0	44	2	2	0
Subtotal (for the Mapping File)			**1**	**2052**		**87**	**330**	**358**	**30**	**65**	**253**	**274**	**473**
Semantic Data Extraction from an RDF Dump File	–	*query()*	17	40	GraphExtractor	17	5	0	30	1	Auto Generation		
Semantic Data Extraction through a SPARQL Endpoint	–	*ExtractTriplesFrom-Endpoint()*	0	100	GraphExtractor	0	5	0	30	0	Auto Generation		
Cleansing	–	*Built-in Petl functions*	15	80	Transformation-OnLiteral	5	12	1	0	1	Auto Generation		
Join	–	*Built-in Petl functions*	1	112	Join-Transformation	1	15	1	0	1	Auto Generation		

Table 2

(Continued)

Tool ⇒		*SETL$_{PROG}$*				*SETL$_{CONSTRUCT}$*				*SETL$_{AUTO}$*			
						NOUC	Components of Each Concept			NOUC	Components of Each Concept		
ETL Task ⇓	Sub construct	Procedures/Data structures	NUEP	NOTC	Task/Operation		Clicks	Selections	NOTC		Clicks	Selections	NOTC
Level Member Generation	–	createDataTripleToFile(), createDataTripleToTriple-Store()	16	75	LevelMember-Generator	16	6	6	0	1	Auto Generation		
Observation Generation	–	createDataTripleToFile(), createDataTripleToTriple-Store()	1	75	Observation-Generator	1	6	6	0	1	Auto Generation		
Loading as RDF Dump	–	insertTriplesIntoTDB(), bulkLoadToTDB()	0	113	Loader	0	1	2	0	0	Auto Generation		
Loading to a Triplestore	–	insertTriplesIntoTDB(), bulkLoadToTDB()	17	153	Loader	17	1	2	0	1	Auto Generation		
Auto ETL Generation	–	–	–	–	CreateETL	1	–	–	–	1	21	16	0
Total (for the Execution Layer Tasks)			**1**	**2807**		**57**	**279**	**142**	**363**	**58**	**21**	**16**	**0**
Grand Total (for the Whole ETL Process)			**1**	**29358**	**245**		**991**	**826**	**2298**	**224**	**656**	**632**	**2378**

Table 3
Comparison between the ETL processes of $SETL_{PROG}$ and PDI for SDW (Reproduced from [44])

Tools	SETL			PDI (Kettle)		
Task	Used tools	Used languages	LOC	Used tools	Used languages	LOC
TBox with MD semantics	Built-in SETL	Python	312	Protege, SETL	Python	312
Ontology Parser	Built-in SETL	Python	2	User Defined Class	Java	77
Semantic Data Extraction through SPARQL endpoint	Built-in SETL	Python, SPARQL	2	Manually extraction using SPARQL endpoint	SPARQL	NA
Semantic Data Extraction from RDF Dump file	Built-in SETL	Python	2	NA	NA	*NA*
Reading CSV/Database	Built-in Petl	Python	2	Drag & Drop	NA	Number of used Activities: 1
Cleansing	Built-in Petl	Python	36	Drag & Drop	NA	Number of used Activities: 19
IRI Generation	Built-in SETL	Python	2	User Defined Class	Java	22
Triple Generation	Built-in SETL	Python	2	User Defined Class	Java	60
External Linking	Built-in SETL	Python	2	NA	NA	NA
Loading as RDF dump	Built-in SETL	Python	1	Drag & Drop	NA	Number of used Activities: 1
Loading to Triple Store	Built-in SETL	Python	1	NA	NA	NA
Total LOC for the complete ETL process			**401**	**471 LOC + 4 NA + 21 Activities**		

requires 12 clicks and 1 selection, and *JoinTransformation* takes 15 clicks and 1 selection.

To create a level member and observation, $SETL_{PROG}$ uses *createDataTripleToFile()* and takes 125 NOTC. The procedure takes all the classes, properties, and blank nodes of a target TBox as input; therefore, the given TBox should be parsed for being used in the procedure. On the other hand, $SETL_{CONSTRUCT}$ uses the *LevelMemberGenerator* and *ObservationGenerator* operations, and each operation requires 1 concept, which takes 6 clicks and 6 selections. $SETL_{PROG}$ provides procedures for either bulk or trickle loading to a file or an endpoint. Both procedures take 113 and 153 NOTC, respectively. For loading RDF triples either to a file or a triple store, $SETL_{CONSTRUCT}$ uses the *Loader* operation, which needs 2 clicks and 2 selections. Therefore, $SETL_{CONSTRUCT}$ reduces NOTC for all transformation and loading tasks by 100%.

$SETL_{AUTO}$ requires only a target TBox and a mapping file to generate ETL data flows through the *CreateETL* interface, which takes only 1 concept composed of 21 clicks and 16 selections. Therefore, other ETL Layer tasks are automatically accomplished internally. In summary, $SETL_{CONSTRUCT}$ uses 92% fewer NOTC than $SETL_{PROG}$, and $SETL_{AUTO}$ further reduces NOUC by another 25%.

8.1.1. Comparison between $SETL_{PROG}$ 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 $SETL_{PROG}$ 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, $SETL_{PROG}$ 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 $SETL_{PROG}$. $SETL_{PROG}$ 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 $SETL_{PROG}$ 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, $SETL_{PROG}$ provides methods to extract semantic data either from a SPARQL endpoint or an RDF dump file batch-wise. In $SETL_{PROG}$, 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. $SETL_{PROG}$ 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 $SETL_{PROG}$. 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, $SETL_{PROG}$ takes 401 Lines of Code (LOC) to run the ETL, where PDI takes $471 LOC + 4 Not\ Applicable(N/A) + 21 Activities$. Thus, $SETL_{PROG}$ 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 $SETL_{PROG}$ uses 14.9% less LOC than PDI, $SETL_{CONSTRUCT}$ uses 92% fewer NOTC than $SETL_{PROG}$, and $SETL_{AUTO}$ further reduces NOUC by another 25%. Combining these results, we see that $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ have significantly better productivity than PDI for building an SDW.

8.2. Development time

We compare the time used by $SETL_{PROG}$, $SETL_{CONSTRUCT}$, and $SETL_{AUTO}$ 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 4

ETL development time (in minutes) required for $SETL_{PROG}$, $SETL_{CONSTRUCT}$, and $SETL_{AUTO}$

Tool	Iteration number	Target TBox definition	Mapping generation	ETL design	Total
$SETL_{PROG}$	1	186	51	85	322
$SETL_{PROG}$	2	146	30	65	241
$SETL_{CONSTRUCT}$	1	97	46	35	178
$SETL_{CONSTRUCT}$	2	58	36	30	124
$SETL_{AUTO}$	1	97	40	2	139
$SETL_{AUTO}$	2	58	34	2	94

$SETL_{PROG}$ took twice as long as $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ to develop a target TBox. In $SETL_{PROG}$, 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()*. $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ create l by typing the input or selecting from the suggested items. Thus, $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ also reduce the risk of making mistakes in an error-prone task, such as creating an ETL. In $SETL_{PROG}$, we typed all the source and target properties to create a mapping dictionary for each source and target concept pair. However, to create mappings, $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ 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, $SETL_{AUTO}$ took less time than $SETL_{CONSTRUCT}$ in mapping generation because we did not need to create mappings for intermediate results.

To create ETL data flows using $SETL_{PROG}$, we had to write scripts for cleansing, extracting, transforming, and loading. $SETL_{CONSTRUCT}$ creates ETL flows using drag-and-drop options. Note that the mappings in $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ use expressions to deal with cleansing and transforming related tasks; however, in $SETL_{PROG}$ we cleansed and transformed the data in the ETL design phase. Hence, $SETL_{PROG}$ took more time in designing ETL compared to $SETL_{CONSTRUCT}$. On the other hand, $SETL_{AUTO}$ creates the ETL data flows automatically from a given mapping file and the target TBox. Therefore, $SETL_{AUTO}$ took only two minutes to create the flows. In short, $SETL_{PROG}$ is a programmatic environment, while $SETL_{CONSTRUCT}$ and $SETL_{AUTO}$ are drag and drop tools. We exemplify this fact by means of Figs 9 and 10, which showcase the creation of the ETL

```
if __name__ == "__main__" :
    datawarehouseTBox.createOntology() #TBox Creation
    query(loalRDFfile='C:/Experiment/recipient.nt', query='select * where { ?a  ?d ?f.}',
            outfilepath='C:/Experiment/temp/Recipient.csv') # Extraction from recipient.nt
    query(loalRDFfile='C:/Experiment/company.nt', query='select * where { ?a  ?d ?f.}',
            outfilepath='C:/Experiment/temp/Company.csv') # Extraction from company.nt
    query(loalRDFfile='C:/Experiment/subsidy.nt', query='select * where { ?a  ?d ?f.}',
            outfilepath='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.outerjoin(recipient, company,
                                key=['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=dict() # creating a Python dictionary for recipient
    mapping_subsidy=dict()  # creating a Python dictionary for subsidy
    mapping_recipient={'hastown':'mdProperty:inCity', 'recipientid':'mdAttribute:recipientid',
            'recipientname':'mdAttribute:recipientname', 'address':'mdAttribute:address',
            'companyid':'mdAttribute:hasCompany','globalType':'mdAttribute:hasGlobalType'}
    mapping_subsidy={'amounteuro':'mdAttribute:amounteuro','year':'mdProperty:year',
            'hasrecipient':'mdProperty:hasrecipientid','source':'mdProperty:source',
            'haspaydate':'mdProperty:haspaydate','budgetline':'mdProperty:budgetline'}
    start=giveSecond(time.localtime())
    file=csv.DictReader(open(r'C:\\Experiment\\temp\\recipient_final.csv',
                        encoding='utf8')) # creating a dictionary from recipient_final.csv
    file.name='Recipient'
    x=createdataTripleToFile(file, sdw, 'recipientid',mapping_recipient, classList,
                             propertyList, 'a') # create RDF triples for Recipient Level
    file=csv.DictReader(open(r'C:\\Experiment\\temp\\subsidy.csv', encoding='utf8'))
    file.name='Subsidy'
    x=createdataTripleToFile(file, sdw, 'subsidyid',mapping_subsidy, classList,
                             propertyList, 'a') # create RDF triples for Subsidy dataset
    start2=giveSecond(time.localtime())
    print("Time to create MD data" + str(start2-start))
```

Fig. 9. The segment of the main method to populate the `sdw:SubsidyMD` dataset and the `sdw:Recipient` using *SETL*$_{PROG}$.

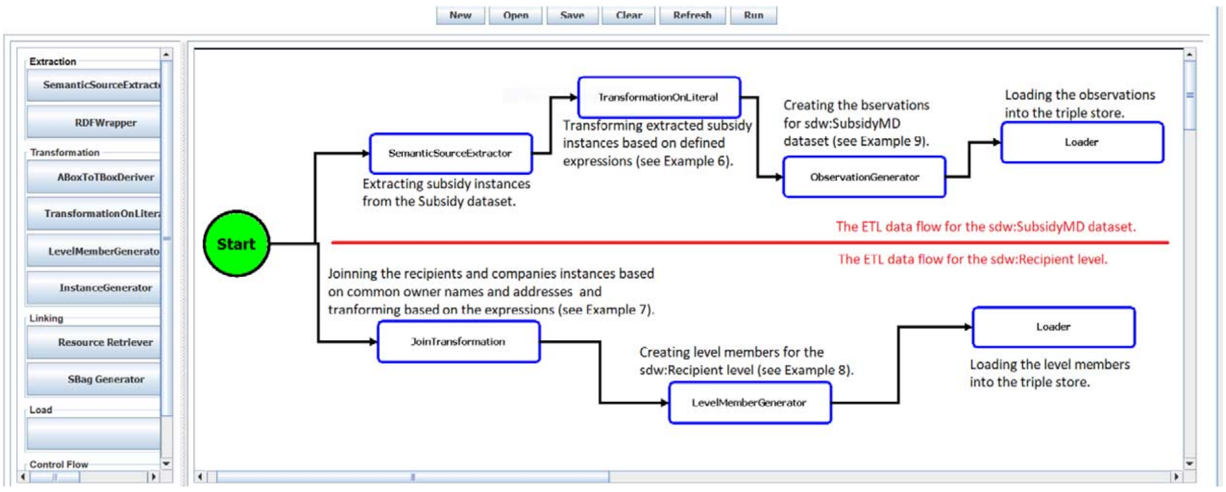

New
Open
Save
Clear
Refresh
Run
Extraction
SemanticSourceExtract
RDFWrapper
Transformation
ABoxToTBoxDeriver
TransformationOnLiter
LevelMemberGenerato
InstanceGenerator
Linking
Resource Retriever
SBag Generator
Load
Control Flow
Start
SemanticSourceExtractor
Extracting subsidy instances from the Subsidy dataset.
TransformationOnLiteral
Transforming extracted subsidy instances based on defined expressions (see Example 6).
Creating the bservations for sdw:SubsidyMD dataset (see Example 9).
ObservationGenerator
Loading the observations into the triple store.
Loader
The ETL data flow for the sdw:SubsidyMD dataset.
The ETL data flow for the sdw:Recipient level.
Joinning the recipients and companies instances based on common owner names and addresses and tranforming based on the expressions (see Example 7).
JoinTransformation
Creating level members for the sdw:Recipient level (see Example 8).
LevelMemberGenerator
Loader
Loading the level members into the triple store.

Fig. 10. The ETL flows to populate the `sdw:SubsidyMD` dataset and the `sdw:Recipient` level using *SETL*$_{CONSTRUCT}$.

data flows for the `sdw:SubsidyMD` dataset and the `sdw:Recipient` level. To make it more readable and understandable, we add comments at the end of the lines of Fig. 9 and in each operation of Fig. 10. In summary, using *SETL*$_{CONSTRUCT}$, the development time is cut in almost half (41% less development time than *SETL*$_{PROG}$); and using *SETL*$_{AUTO}$, it is cut by another 27%.

Table 5

ETL execution time (in minutes) required for each sub-phase of the ETL processes created using $SETL_{PROG}$ and $SETL_{CONSTRUCT}$

Performance metrics	Systems	Extraction and traditional transformation	Semantic transformation	Loading	Total processing time
Processing time (in minutes)	$SETL_{PROG}$	33	17.86	21	71.86
	$SETL_{CONSTRUCT}$	43.05	39.42	19	101.47
Input size	$SETL_{PROG}$	6.2 GB (Jena TDB) + 6.1 GB (N-Triples)	496 MB (CSV)	4.1 GB (N-Triples)	
	$SETL_{CONSTRUCT}$	6.2 GB (Jena TDB) + 6.1 GB (N-Triples)	6.270 GB (N-Triples)	4.1 GB (N-Triples)	
Output size	$SETL_{PROG}$	490 MB (CSV)	4.1 GB (N-Tripels)	3.7 GB (Jena TDB)	
	$SETL_{CONSTRUCT}$	6.270 GB (N-Triples)	4.1 GB (N-Triples)	3.7 GB (Jena TDB)	

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-

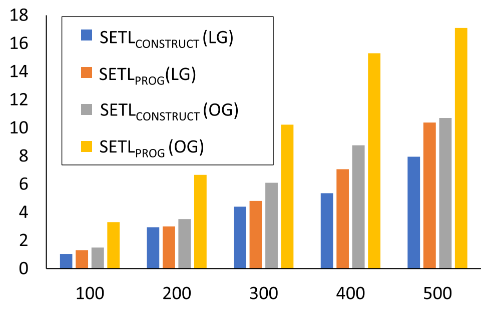


Fig. 11. Comparison of $SETL_{CONSTRUCT}$ and $SETL_{PROG}$ for semantic transformation. Here, LG and OG stand for *LevelMemberGenerator* and *ObservationGenerator*.

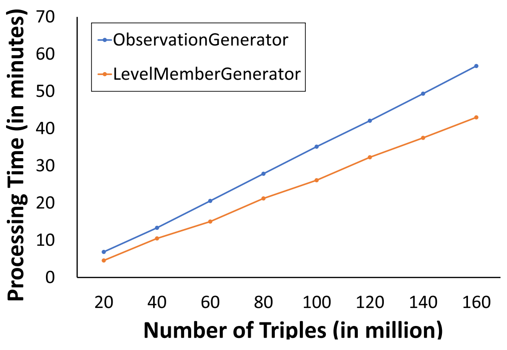


Fig. 12. Scalability of *LevelMemberGenerator* and *Observation-Generator*.

eration and the $SETL_{PROG}$ procedure increases with the size of the input. In order to guarantee scalability when using the Jena library, a $SETL_{CONSTRUCT}$ 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 *LevelMemberGenerator* and *ObservationGenerator* operations with the increasing number of triples. We show the scalability of the *LevelMemberGenerator* and *Observation-Generator* 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. $SETL_{CONSTRUCT}$ takes less time in loading than $SETL_{PROG}$ because $SETL_{PROG}$ uses the Jena TDB loader command to load the data while $SETL_{CONSTRUCT}$ programmatically load the data using the Jena API's method.

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

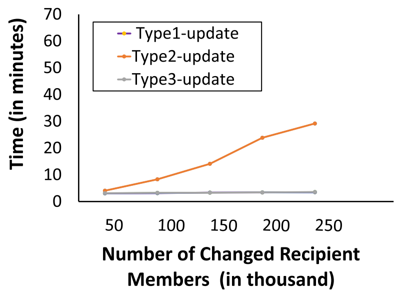
Besides the differences of the performances already explained in Table 5, $SETL_{CONSTRUCT}$ also includes an operation to update the members of the SDW levels, which is not included in $SETL_{PROG}$. Since the ETL process for our use case SDW did not include that operation, we scrutinize the performance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

Performance analysis of UpdateLevel operation Figure 13 shows the performance of the *UpdateLevel* operation. To evaluate this operation, we consider two levels: `mdProperty:Recipient` and `mdProperty:City`. We consider these two levels because `mdProperty:Recipient` is the largest level (in terms of size) in our use case, and `mdProperty:City` is the immediate upper level of `mdProperty:Recipient` in the `mdStructure:Address` hierarchy of the dimension `mdProperty:Beneficiary`. Therefore, we can record how the changes in cities propagate to recipients, especially in Type2-update. The `sdw:Recipient` level is the lowest granularity level in the `mdStructure:Address` hierarchy; therefore, changes in a recipient (i.e., a member of `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 Type-3 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.

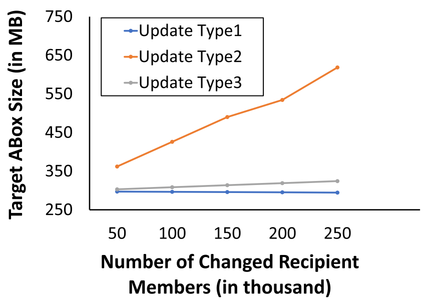
Figure 13c shows the processing time when increasing the number of updated cities. Since `sdw:City` is the immediate upper level of `sdw:Recipient` in the `mdStructure:address` hierarchy, to reflect a changed city in the target ABox, a Type2-update creates new versions for itself as well as for the recipients 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.

8.3.1. *Comparison between $SETL_{PROG}$ and PDI in processing `sdw:Subsidy`*

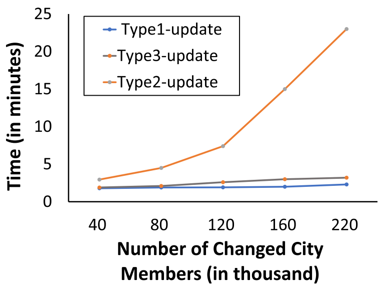
In [44], we compare the performance of $SETL_{PROG}$ with a non-semantic data integration tool, PDI. We populate the SDW for `sdw:Subsidy` concept using PDI. To run the process PDI takes 1903 s. On other hand, SETL takes only 1644 s. Thus, SETL is 259 s (13.5%) faster than PDI. PDI is much slower because



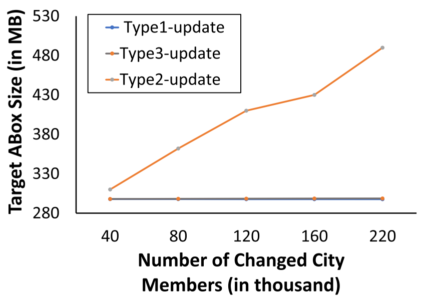
(a) Processing time of increasing changed recipient members.



(b) Target size of increasing changed recipient members.



(c) Processing time of increasing changed city members.



(d) Target size of increasing changed city members.

Fig. 13. Performance of *UpdateLevel* based on processing time and target ABox size with the changing of the `sdw:Recipient` and `sdw:City` members.

almost everything MD-related must be embedded as UDFs or equivalent elements.

In overall, we conclude that $SETL_{PROG}$ is better in performance compared to PDI and $SETL_{CONSTRUCT}$. However, $SETL_{CONSTRUCT}$ is a more correct and scalable framework. On top of that, $SETL_{CONSTRUCT}$ 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/1qgtijnXyLtOabnSR58LyoP3eqOizGZdZ/view?usp=sharing

Table 6

Summary of the expert interviews

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

posed 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 ($SETL_{PROG}$) 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 $SETL_{PROG}$ 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 $SETL_{PROG}$ 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 $SETL_{PROG}$ 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 $SETL_{PROG}$. 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 $SETL_{CONSTRUCT}$ based on the high-level constructs proposed in this paper. We also extended it to enable automatic ETL execution flows generation (and named it $SETL_{AUTO}$). The experiment shows that 1) $SETL_{CONSTRUCT}$ uses 92% fewer NOTC than $SETL_{PROG}$, and $SETL_{AUTO}$ further reduces NOUC by another 25%; 2) using$SETL_{CONSTRUCT}$, the development time is almost cut in half compared to $SETL_{PROG}$, and is cut by another 27% using $SETL_{AUTO}$; 3) $SETL_{CONSTRUCT}$ is scalable and has similar performance compared to $SETL_{PROG}$.

$SETL_{CONSTRUCT}$ 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 *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 on virtual semantic DWs, similar to the ones developed for virtual integration of data cubes and XML data [48,49,63]. Another interesting work will be to apply this layer-based integration process in a Big Data and Data Lake environment. The metadata produced in the Definition Layer can be used as basis to develop advanced catalog features for Data Lakes. We will also introduce user-defined functions (UDFs) and other relevant ETL operations to make the approach fit in the Big Data context, as suggested by the two interviewed experts. Furthermore, we plan to investigate how to integrate provenance information into the process [2]. Another aspect of future work is the support of spatial data [23,24].

Acknowledgements

This research is partially funded by the European Commission through the Erasmus Mundus Joint Doctorate Information Technologies for Business Intelligence (EM IT4BI-DC), the Poul Due Jensen Foundation, and the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B.

Appendix

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

EXECUTEQUERY*(Q, G, outputHeader)* This function provides similar functionality as SPARQL SELECT queries.

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:[18]

1. An RDF triple pattern t_p is a query pattern Q. A t_p[19] 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 $(Q_1 \text{ AND } Q_2)$, $(Q_1 OPT Q_2)$, 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 $(<, \geqslant, \leqslant, >)$, logical connectivities $(\neg, \vee, \wedge)$, unary predicates like bound, isBlank, and isIRI plus other features described in [25].

G is an RDF graph over which Q is evaluated and *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].

Semantics: For the query pattern Q, let μ be a partial function that maps var(Q) to T, i.e., $\mu : \text{var}(Q) \rightarrow T$. The domain of μ, denoted by dom(μ), is the set of variables occurring in Q for which μ 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 μ. 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., $[\![Q]\!]_G = [\![t_p]\!]_G = \{\mu | \text{dom}(\mu) = \text{var}(t_p) \wedge \mu(t_p) \in G\}$.

[18]We follow the same syntax and semantics used in [50] for a query pattern.

[19]To make it easily distinguishable, here we use comma to separate the components of a triple pattern and an RDF triple.

2. If Q is $(Q_1\ AND\ Q_2)$, then $[\![Q]\!]_G$ is the natural join of $[\![Q_1]\!]_G$ and $[\![Q_2]\!]_G$, i.e., $[\![Q]\!]_G = [\![Q_1]\!]_G \bowtie [\![Q_2]\!]_G = \{\mu_1 \cup \mu_2 | \mu_1 \in [\![Q_1]\!]_G, \mu_2 \in [\![Q_2]\!]_G, \mu_1$ and μ_2 are compatible mappings, i.e., $\forall ?x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2), \mu_1(?x) = \mu_2(?x)\}$.
3. If Q is $(Q_1\ OPT\ Q_2)$, then $[\![Q]\!]_G$ is the left outer join of $[\![Q_1]\!]_G$ and $[\![Q_2]\!]_G$, i.e., $[\![Q]\!]_G = [\![Q_1]\!]_G ⟕ [\![Q_2]\!]_G = ([\![Q_1]\!]_G \bowtie [\![Q_2]\!]_G) \cup ([\![Q_1]\!]_G \setminus ([\![Q_2]\!]_G)$.
4. If Q is $(Q_1$ UNION $Q_2)$, then $[\![Q]\!]_G = [\![Q_1]\!]_G \cup [\![Q_2]\!]_G = \{\mu | \mu \in [\![Q_1]\!]_G$ or $\mu \in [\![Q_2]\!]_G\}$.
5. If $Q = (Q_1$ FILTER $F_c)$, then $[\![Q]\!]_G = \{\mu | \mu \in [\![Q_1]\!]_G \wedge \mu \models F_c\}$, where a mapping μ 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 μ.

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.

Input Parameters: *sTBox* is a TBox and *exps* is a set of expressions.

Semantics: For each *exp* ∈ *exps*, this function computes the intersection of the set of IRIs 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 IRIs used in *exp* and $\mathcal{P}(t)$ is defined in Equation (2). The semantic is defined as follows:

$$\text{GETPROPERTIESFROMEXPRESSIONS}(sTBox, exps) = \bigcup_{exp \in exps} returnIRIs(exp) \cap \mathcal{P}(sTBox).$$

This function returns a set of properties.

VALIDATEEXPRESSIONS*(exps, Q, flag)* The source expressions/properties (defined by `map:source4TargetPropertyValue`) in property-mappings contains properties. However, the *outputHeader* parameter of EXECUTEQUERY() allows only query variables. Therefore to extract data from an RDF graph using 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 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.

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*.

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, 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.

MAPPEDSOURCEINSTANCES*(sc, sTBox, sABox, propertyMappings)* This function returns a dictionary describing the instances of a source concept.

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

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:

$$\textsc{mappedSourceInstances}(sc, sTBox, sABox, propertyMapping) = dictionary\Big(\textsc{executeQuery}\Big(\big(?i, \texttt{rdf:}type^{[20]}, sc\big)\ AND\ (?i, ?p, ?v) FILTER \Big(?p \in \textsc{getPropertiesFromExpressions}\Big(sTBox, mappedExpressions(sc, propertyMappings)\Big)\Big)\Big), sABox, (?i, ?p, ?v)\Big)\Big).$$

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 $i \in ?i$, it creates a set of (p_i, v_i) pairs, where p_i is a property of i and v_i is the value for p_i. 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]; $\mathcal{C}(T)$ and $\mathcal{P}(T)$ 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*.

$$generateIRI(sIRI, value, tType, tTBox, iriGraph)$$
$$= \begin{cases} lookup(sIRI, iriGraph) & \text{if } lookup(sIRI, iriGraph) != NULL \\ concat(tType, \text{“\#”}, validate(value)) & \text{if } lookup(value, iriGraph) = NULL \ \wedge\ tType \in (\mathcal{C}(tTBox) \cup \mathcal{P}(tTBox)) \\ concat(prefix(tTBox), \text{“\#”}, validate(value)) & \text{if } lookup(sIRI, iriGraph) = NULL \ \wedge\ tType \notin (\mathcal{C}(tTBox) \cup \mathcal{P}(tTBox)) \end{cases} \quad (3)$$

This function returns an IRI.

TUPLESTOTRIPLES*(T, tc, Q, propertyMappings, (i,* $exp_1, \ldots, exp_n$*))* 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 $(i, exp_1, \ldots, exp_n)$ 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 (exp_i) and the value of the expression ($\text{val}(exp_i)$) represent the corresponding target property and its value of the instance. Here, $\text{val}(exp_i)$ 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, VALIDATEEXPRESSIONS*(exp*$_i$*, 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*.

$$\textsc{tuplesToTriples}\big(T, tc, Q, propertyMappings, (i, exp_1, \ldots, exp_n)\big)$$
$$= \bigcup_{(\text{val}(i), \text{val}(exp_1), \ldots, \text{val}(exp_n)) \in T} \Bigg\{ \Big\{\big(\text{val}(i), \texttt{rdf:type}, tc\big)\Big\}$$
$$\cup \bigcup_{p=1}^{n} \Big\{\big(val(i), return\big(\textsc{validateExpressions}$$

[20]If the source construct is either a QB level property or dataset, then `rdf:type` is replaced by either `qb4o:memberOf` or `qb4o:dataset`.

$$\times (exp_p, Q, 0), propertyMappings\big),$$

$$val(exp_i))\}\Bigg\} \tag{4}$$

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 EXECUTEQUERY(). G is an RDF graph over which Q is evaluated. The parameter *outputPattern* is a set of triple patterns from $(I \cup B \cup var(Q)) \times (I \cup var(Q) \times (T \cup var(Q))$, where $var(Q) \subseteq V$ is the set of query variables occurring in Q. Here, I, B, T, and V are the sets defined in EXECUTEQUERY(). 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 (μ), 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.

$$GraphExtractor(Q, G, outputPattern, tABox) = \bigcup_{t_h \in outputPattern} \{\mu(t_h) | \mu \in [\![Q]\!]_G \wedge \mu(t_h) \text{ is a well-formed RDF triple}\}. \tag{5}$$

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: *sConstruct* and *tConstruct* are a source and target TBox construct, *sTBox* and *sABox* are the source TBox and ABox, *propertyMappings* is a set of property-mappings, and *tABox* is the output location.

Semantics: First, we retrieve the instances of *sConstruct ins(c)* from *sABox* using the EXECUTEQUERY() function, which is formally described as follows:

ins(c)=EXECUTEQUERY*(q(c), sABox,(?i,* VALIDATEEXPRESSIONS*(list(cElements),q(c),1)).*

Here:

- *c = sConstruct.*
- *q(c)=(((?i,* `rdf:type`*, c) AND (?i,?p,?v)) FILTER (?p ∈ cProperties))* is a query pattern.
- *cProperties=* GETPROPERTIESFROMEXPRESSIONS*(sTBox, cElements)* is the set of source properties used in source expressions defined in *propertyMappings*.
- *cElements* = EXECUTEQUERY*((?pm,* `map:source4TargetPropertyValue`*, ?sp), propertyMappings, ?sp)* is the set of source expressions in *propertyMappings* defined by `map:source4TargetPropertyValue`.
- VALIDATEEXPRESSIONS*(list(cElements),q(c),1)* replaces the source properties used in *cElements* with the corresponding query variables from *q(c)* as *outputHeader* parameter EXECUTEQUERY() does not allows 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 $ins(c)$ into equivalent RDF triples to get *tABox*, i.e.,

$$output = \text{TUPLESTOTRIPLES}(ins(c), tConstruct, q(c), (?i, exp_1, \ldots, exp_n)).$$

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: *sConstruct* and *tConstruct* are a source and target[21] TBox construct, *sTBox* and *tTBox* are the source and target TBoxes; *sABox* and *tABox* are the source and target ABoxes; *comProp* is a set

[21]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 *propertyMapping* 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:

$$
\begin{aligned}
&ins(tc, sc)\\
&\quad = \text{EXECUTEQUERY}((q(tc)OPTq(sc)),\\
&\qquad union(extractIns(tc, tABox), extractIns(sc,\\
&\qquad sABox)), (?i,\\
&\qquad \text{VALIDATEEXPRESSIONS}(scElements,\\
&\qquad (q(tc)OPTq(sc)), 1)).
\end{aligned}
$$

Here:

- $tc = tConstruct$ and $sc = sConstruct$.
- $q(tc) = ((?i_{tc}, \texttt{rdf:type}, tc)$ AND $(?i_{tc}, ?p_{tc}, ?v_{tc})tp_{com}(?i_{tc}, \text{“}target'', comProp)$ FILTER$(?p_{tc} \in tcProperties))$ is a query pattern.
  * $tp_{com}(?i_{tc}, target, comProp)$ is a function that joins a triple pattern (i.e., $AND\ (?i_{tc}, scom_i, ?scom_i))$ in $q(tc)$ for each pair $(scom_i, tcom_i) \in comProp$.
  * *tcProperties* = GETPROPERTIESFROMEXPRESSIONS*(tTBox, scElements)* represents the set of source properties used in source expressions.
- $q(sc) = ((?i_{sc}, \texttt{rdf:type}, sc)$ AND $(?i_{sc}, ?p_{sc}, ?v_{sc})sp_{com}(?i_{sc}, \text{“source''}, comProp)$ FILTER$(?p_{sc} \in scProperties))$ is the query pattern.
  * $sp_{com}(?i_{sc}, \text{“}source\text{”}, comProp)$ is a function that joins a triple pattern (i.e., $AND\ (?i_{sc}, tcom_i, ?scom_i))$ in $q(sc)$ for each pair $(scom_i, tcom_i) \in comProp$.
  * *scProperties* = GETPROPERTIESFROMEXPRESSIONS*(sTBox, scElements)* represents the set of source properties used in source expressions.
- *scElements* = EXECUTEQUERY*((?pm,* `map:source4TargetPropertyValue`*, ?sp), propertyMappings, ?sp)* is the set of source expressions in *propertyMappings* defined by `map:source4TargetPropertyValue`.
- *extractInstance(c, abox)* = EXECUTEQUERY*(((?i, rdf:type, c) 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*.
- $union(s1, s2)$ returns the set union of two given sets $s1$ and $s2$.
- VALIDATEEXPRESSIONS*(list(scElements),q(sc, tc),1)* replaces the source properties used in *scElements* with the corresponding query variables from *q(tc,sc)* as the *outputHeader* parameter of 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 *ins*(*sc*, *tc*) into equivalent RDF triples to get the transformed *tABox*, i.e.,

$$
\begin{aligned}
&output\\
&\quad = \text{TUPLESTOTRIPLES}(ins(tc, sc), tConstruct,\\
&\qquad q(tc, sc), (?i, exp_1, \ldots, exp_n)).
\end{aligned}
$$

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

LevelMemberGenerator(sConstruct, level, sTBox, sABox, tTBox, iriValue, iriGraph, propertyMappings, tABox) The objective of this operation is to create QB4OLAP-compliant level members.

Input Parameters: *sConstruct* is the source construct, *level* is a target level, *sTBox* and *sABox* are the source TBox and ABox, *iriValue* is the rule of creating level members' IRIs, *iriGraph* is the IRI graph, *proeprtyMappings* is a set of property-mappings, and *tABox* is the output location.

Semantics: First, we retrieve the instances of *sConstruct* with their properties and values, i.e.,

$$
\begin{aligned}
Ins = {}& \text{MAPPEDSOURCEINSTANCES}(sConstruct,\\
& sTBox, sABox, propertyMappings) \qquad (6)
\end{aligned}
$$

To enrich *level* with the dictionary *Ins*, we define a set of triples *LM* by taking the union of *IdentityTriples* (for each *(i,pv)* pair in *Ins*) and the union of *DescriptionTriples* (for each (p_i, v_i) pair in *pv*). Equation (7) defines *LM*.

$$LM = \bigcup_{(i,pv)\in Ins} \Big(IdentityTriples \cup \bigcup_{(p_i,v_i)\in pv} DescriptionTriple \Big) \quad (7)$$

Here:

– IdentityTriples

$$= \{(lmIRI, \texttt{rdf:type}, \texttt{qb4o:LevelMember}), (lmIRI, \texttt{qb4o:memberOf}, level)\} \quad (8)$$

– lmIRI

$$= \begin{cases} i & \text{if } iriValue = \text{“}sameAsSourceIRI\text{”} \\ \textsc{generateIRI}(i, resolve(i, pv, iriValue), range(level, tBox, iriGraph), tBox) & \text{if } range(level, tBox) != NULL \\ \textsc{generateIRI}(i, resolve(i, pv, iriValue), level, tBox, iriGraph) & \text{if } range(level, tBox) = NULL \end{cases} \quad (9)$$

– DescriptionTriple

$$= \begin{cases} \{(lmIRI, return(p_i, propertyMappings), \textsc{generateIRI}(v_i, iriValue(v_i), range((return(v_i, propertyMappings), tBox), tBox, iriGraph))\} & \text{if } targetType(return(v_i, propertyMappings), tBox) \in \{rollupProperty, ObjectProperty\} \\ \{(lmIRI, return(p_i, propertyMappings), v_i)\} & \text{if } targetType(p_i, propertyMappings, tBox) = levelAttribute \end{cases} \quad (10)$$

Each instance of *sConstruct* is a `qb4o:LevelMember` and a member of *level*; therefore, for each instance of *sConstruct*, *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 *iriValue* is “sameAsSourceIRI” then the function returns the source IRI; otherwise it resolves value of *iriValue* using the *resolve(i, pv, iriValue)* function – this function returns either the value of a property/expression or next incremental value –, 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, $range(level, tBox)$ returns the range of *level* from *tBox*.

Equation (10) creates a triple for each (p_i, v_i) pair of *Ins* (in Equation (6)). If p_i corresponds to either an object property or a rollup property in *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, $return(x, \text{mappings})$ function returns the target construct that is mapped to x from the set of property-mappings mappings; $targetType(x, tBox)$ returns the type of a target construct x from *tBox*; and $iriValue(v)$ 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 *LM* and the operation stores the output in *tABox*.

ObservationGenerator(sConstruct, dataset, sTBox, sABox, tTBox iriValue, iriGraph, propertyMappings, tABox) This operation creates QB4OLAP-compliant observations from the source data.

Input Parameters: *sConstruct* is the source construct, *dataset* is a target QB dataset, *sTBox* and *sABox* are the source TBox and ABox, *iriValue* is the rule of creating level members' IRIs, *iriGraph* is the IRI graph, *proeprtyMappings* is a set of property-mappings, and *tABox* is the output location.

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

$$\mathit{IdentityTriples} = \big\{(\mathit{oIRI}, \texttt{rdf:type}, \texttt{qb:Observation}), (\mathit{oIRI}, \texttt{qb:dataset}, \mathit{dataset})\big\} \tag{11}$$

$$\mathit{oIRI} = \begin{cases} i & \text{if } \mathit{iriValue} = \text{“}\mathit{sameAs\ SourceIRI}\text{”} \\ \textsc{generate}\mathrm{IRI}(i, \mathit{resolve}(i, \mathit{pv}, \mathit{iriValue}), \mathit{dataset}, \mathit{tTBox}, \mathit{iriGraph}) & \mathit{otherwise} \end{cases} \tag{12}$$

$$\mathit{DescriptionTriple} = \begin{cases} \{(\mathit{oIRI}, \mathit{return}(p_i, \mathit{propertyMappings}), \mathit{lmIRI}_o)\} & \text{if } \mathit{targetType}(\mathit{return}(p_i, \mathit{propertyMappings}), \mathit{tTBox}) = \mathit{LevelProperty} \\ \{(\mathit{oIRI}, \mathit{return}(p_i, \mathit{propertyMappings}), v_i)\} & \text{if } \mathit{targetType}(\mathit{return}(p_i, \mathit{propertyMappings}), \mathit{tTBox}) = \mathit{MeasureProperty} \end{cases} \tag{13}$$

$$\mathit{lmIRI}_o = \begin{cases} \textsc{generate}\mathrm{IRI}(v_i, \mathit{iriValue}(v_i), \mathit{range}(\mathit{return}(p_i, \mathit{property{-}Mappings}), \mathit{tTBox}), \mathit{tTBox}, \mathit{iriGraph}) & \text{if } \mathit{range}(\mathit{return}(p_i, \mathit{propertyMappings}), \mathit{tTBox}))! = \mathit{NULL} \\ \mathit{generateIRI}(v_i, \mathit{iriValue}(v_i), \mathit{return}(p_i, \mathit{propertyMappings}), \mathit{tTBox}, \mathit{iriGraph}) & \text{if } \mathit{range}(\mathit{return}(p_i, \mathit{propertyMappings}), \mathit{tTBox})) = \mathit{NULL} \end{cases} \tag{14}$$

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

The output of this operation is *OB*, which will be stored in *tABox*.

<u>*ChangedDataCapture(nABox, oABox, flag)*</u> This operation triggers either the SDW evolution (i.e., enriched with new instances) or update (i.e., reflect the changes in the existing instances).

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

Semantics: Here, *nABox* and *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 *nABox* and *oABox* using the EXECUTEQUERY() function, shown as follows:

$$\mathit{Ins}_{\mathit{nABox}} = \textsc{executeQuery}((?i, \texttt{rdf:type}, ?v), \mathit{nABox}, ?i)$$

$$\mathit{Ins}_{\mathit{oABox}} = \textsc{executeQuery}((?i, \texttt{rdf:type}, ?v), \mathit{oABox}, ?i)$$

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

$$\mathit{InsDes}_{\mathit{new}} = \bigcup_{i \in \mathit{Ins}_{\mathit{new}}} \textsc{executeQuery}\big((?s, ?p, ?v)\ \mathit{FILTER}(?s = i)\big), \mathit{nABox}, (?s, ?p, ?v)) \tag{15}$$

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

$$\mathit{ChangedTriples} = \mathit{InsDes}_{\mathit{old}} - \mathit{oldABox}.$$

If $(\mathit{flag} = 0)$ then the output is $\mathit{InsDes}_{\mathit{new}}$, else *ChangedTriples*. The output overwrites *oABox*.

<u>*updateLevel(level, updatedTriples, sABox, tTBox, tABox, propertyMappings, iriGraph)*</u> 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*, *tTBox* 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:

$$sABox = (instance, property, oldVal)$$

$$updatedTriples = (instance, property, newVal)$$

This operation updates *tABox* 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).

$$tABox = tABox - DeleteTriples \tag{16}$$

$$tABox = tABox \cup InsertTriples \tag{17}$$

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

$$NewOldValues = updatedTriples \bowtie sABox \tag{18}$$

We define *DeleteTriples* and *InsertTriples* as

$$DeleteTriples = \bigcup_{(i,p,nV)\in updatedTriples} del(i, p, nV) \tag{19}$$

$$InsertTriples = \bigcup_{(i,p,nV,oV)\in NewOldValues} in(i, p, nV, oV) \tag{20}$$

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

$$updateType = \big(updateType\big(return(p, propertyMappings), tTBox\big)\big) \tag{21}$$

In the following, we describe how the functions, $del(i, p, nV)$ and $in(i, p, nV, oV)$ are defined for different values of *updateType*.

If *updateType* = Type-1 update

If *updateType* is *Type-1 update*, the new value of the property will replace the old one. Therefore, the triple holding the old value for the property will be deleted from *tABox* and a new triple with the updated value of the property will be inserted into *tABox*. Equation (22) defines $del(i, p, nV)$ for the Type-1 update. For an instance i in the source, the existing equivalent IRI in the target can be retrieved from the IRI graph *iriGraph*, i.e., $IRI_i = lookup(i, iriGraph)$. *return(p, propertyMappings)* in Equation (22) returns the target level attribute that is mapped to p from *propertyMappings*.

$$\begin{aligned} del(i, p, nV) &= \textsc{executeQuery}\big(\big((?i, ?p, ?\,\mathrm{val}) \\ &\quad FILTER\big(?i = IRI_i \&\& ?p = return(p, \\ &\quad property\ Mappings)\big)\big), tABox, (?i ?p ?\,\mathrm{val})\big) \end{aligned} \tag{22}$$

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(org_str, search_pattern, replace_pattern)* function updates the string *org_str* by replacing the substring *search_pattern* with *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.

$$\begin{aligned} in(i, p, nV, oV) &= \big\{(IRI_i, return(p, mapping), \\ &\quad replace\big(targetValue(i, p), oV, nV\big)\big\} \end{aligned} \tag{23}$$

where,

$$targetValue(i, p) = \text{EXECUTEQUERY}\big((IRI_i, return(p, mapping), ?v), tABox, ?v\big) \tag{24}$$

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 *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 *property*, *concat*(*property*, "*_oldValue*") creates a property to contain the old value of the *property* by concatenating the "*_oldValue*" with *property*. The function $targetValue(i, p)$ returns the objects of triple whose subject and predicate correspond to i and p, defined in Equation (24).

$$del(i, p, nV) = \text{EXECUTEQUERY}\Big(\big((?i, ?p, ?\text{val}) \; FILTER\big(?i = IRI_i \&\& ?p \in \big(return(p, propertyMappings), concat\big(return(p, propertyMappings), \text{“\_oldValue”}\big)\big)\big), tABox, (?i, ?p, ?\text{val})\big)\Big) \tag{25}$$

$$in(i, p, nV, oV) = \big\{(IRI_i, return(p, mapping), replace\big(targetValue(i, p), oV, nV\big), \big(IRI_i, concat\big(return(p, propertyMappings), \text{“\_oldValue”}\big), targetValue(i, p)\big)\big\} \tag{26}$$

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 `type2:toDate`) and status (defined by `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 *tABox*. Equation (27) defines $del(i, p, nV)$ for Type-2 update. The first operand of the union in Equation (27) retrieves the expired triples of IRI_i (the level member corresponding to i) from *tABox*. As the level of 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 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) $getExpiredTriplesAll(IRI_i)$ returns the set of expired triples of the members of all lower levels referring to IRI_i (described in Equation (28)). The function $getTriplesImmediate(IRI_i)$ returns the set of expired triples of the members of immediate child level referring to IRI_i (described in Equation (29)).

$$del(i, p, nV) = \text{EXECUTEQUERY}\Big(\big((IRI_i, ?p, ?\text{val}) \; FILTER\big(?p \in (\texttt{type2:toDate}, \texttt{type2:status})\big)\big), tABox, (IRI_i, ?p, ?\text{val})\Big) \cup getExpiredTriplesAll(IRI_i) \tag{27}$$

$$getExpiredTriplesAll(IRI_i) = \bigcup_{(i_c\ p\ v) \in getExpiredTriplesImmediate(IRI_i)} getExpiredTriplesAll(i_c) \tag{28}$$

$$getExpiredTriplesImmediate(IRI_i) = \text{EXECUTEQUERY}\big((?i_c, ?p, IRI_i) \; AND \; (?i_c, \texttt{rdf:type}, \texttt{qb4o:LevelMember}) \; AND \; (?i_c, ?p, ?v) \; FILTER\big(?p \in (\texttt{type2:toDate}, \texttt{type2:status})\big)\big), tABox, (?i_c, ?p, ?\text{val})\big) \tag{29}$$

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 *in(i,p,nV,oV)* for Type-2 update. The IRI for the new version of IRI_i is $newIRI_i$ = *updateIRI*(IRI_i*, iriGraph)*. The *updateIRI(iri, iriGraph)* function updates the existing IRI *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 replacing the old value with the new value, the predicate is the target level attribute equivalent to p, and the subject is $newIRI_i$ (first operand of the union operation in Equation (30)). The call of *update2Insert(*iri_o*,* iri_n*,cp)* in Equation (30) returns the triples required to reflect the changes in *tABox*. Here, iri_o and iri_n are the old and new version (IRI) of a level member and *cp* is the predicate of the updated triple (i.e., the property whose value has been changed). The values of `type2:status` for new and existing versions are set to "Current" and "Expired", respectively. The validity interval of existing version is ended on *sysdate()-1* and the validity interval of new version is from *sysdate()* to "9999-12-31". The *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 `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 iri_o with unchanged properties using *retrieveTriples(*iri_o*, cp)* which returns all triples of iri_o except the triples with the properties *cp* (changed property), `type2:toDate`, `type2:fromDate`, and `type2:status` (described in Equation (32)). We replace the IRI of old version with the new one using the *replace()* function. To propagate the current version to its descendants (i.e., the level members those are associated to iri_o); the function *updateAssociates(*iri_o*,* 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 *getAssociates(*iri_o*)* returns the set of triples that describe the members that are connected to iri_o with a property. For each member connected to iri_o, we create a new version of it and also recursively update the members dependent on it using Equation (35).

$$\begin{aligned} &in(i, p, nV, oV)\\ &\quad = \big\{(newIRI_i, return(p, propertyMappings),\\ &\qquad replace\big(targetValue(i, p), oV, nV\big)\big\}\\ &\qquad \cup\, update2Insert\big(IRI_i, newIRI_i, return(p,\\ &\qquad propertyMappings)\big) \qquad (30)\end{aligned}$$

$$\begin{aligned} &update2Insert(iri_o, iri_n, cp)\\ &\quad = \big\{\big(iri_o, \mathtt{type2:status}, \text{“}Expired\text{”}\big),\\ &\qquad \big(iri_o, \mathtt{type2:toDate}, sysdate() - 1\big),\\ &\qquad \big(iri_n, \mathtt{type2:fromDate}, sysdate()\big),\\ &\qquad \big(iri_n, \mathtt{type2:toDate}, \text{“}9999 - 12 - 31\text{”}\big),\\ &\qquad \big(iri_n, \mathtt{type2:status}, \text{“}Current\text{”}\big)\big\}\\ &\qquad \cup\, replace\big(retrieveTriples(iri_o, cp), iri_o, iri_n)\big)\\ &\qquad \cup\, updateAssociates(iri_o, iri_n) \qquad (31)\end{aligned}$$

$$\begin{aligned} &retrieveTriples(lm, cp)\\ &\quad = \text{EXECUTEQUERY}\big(\big((lm,\\ &\qquad ?p, ?v) FILTER\big(?p \notin (cp, \mathtt{type2:toDate},\\ &\qquad \mathtt{type2:fromDate}, \mathtt{type2:status})\big)\big),\\ &\qquad tABox, (lm, ?p, ?v)\big) \qquad (32)\end{aligned}$$

$$\begin{aligned} &updateAssociates(iri_o, iri_n)\\ &\quad = \bigcup_{(i_c, p) \in getAssociates(iri_o)} recursiveUpdate(i_c, p) \qquad (33)\end{aligned}$$

$$\begin{aligned} &getAssociates(iri_o)\\ &\quad = \text{EXECUTEQUERY}\\ &\qquad \big(\big((?i_c, ?p, iri_o)\ AND\ (?i_c, \mathtt{rdf:type}, \mathtt{qb4o:}\\ &\qquad \mathtt{LevelMember})\big), tABox, (?i_c, ?p)\big) \qquad (34)\end{aligned}$$

$$\begin{aligned} &recursiveUpdate(i_c, p)\\ &\quad = \big\{\big(updateIRI(i_c, iriGraph), p, i_c\big)\big\} \cup\\ &\qquad update2Insert\big(i_c, updateIRI(i_c, iriGraph), p\big) \qquad (35)\end{aligned}$$

This operation updates the target ABox, *tABox*.

We refer to [44] for the semantics of the *ExternalLinking* operation and to [26] for the semantics of the *MaterializeInference* operation.

References

[1] A. Abelló, O. Romero, T.B. Pedersen, R. Berlanga, V. Nebot, M.J. Aramburu and A. Simitsis, Using semantic web technologies for exploratory OLAP: A survey, *IEEE transactions on knowledge and data engineering* **27**(2) (2014), 571–588. doi:10.1109/TKDE.2014.2330822.

[2] K. Ahlstrøm, K. Hose and T.B. Pedersen, Towards answering provenance-enabled SPARQL queries over RDF data cubes, in: *Joint International Semantic Technology Conference*, Springer, 2016, pp. 186–203. doi:10.1007/978-3-319-50112-3_14.

[3] A.B. Andersen, N. Gür, K. Hose, K.A. Jakobsen and T.B. Pedersen, Publishing Danish agricultural government data as semantic web data, in: *Joint International Semantic Technology Conference*, Springer, 2014, pp. 178–186. doi:10.1007/978-3-319-15615-6_13.

[4] F. Baader, D. Calvanese, D. McGuinness, P. Patel-Schneider and D. Nardi, *The Description Logic Handbook: Theory, Implementation and Applications*, Cambridge university press, 2003.

[5] S.K. Bansal, Towards a Semantic Extract-Transform-Load (ETL) framework for big data integration, in: *Big Data*, IEEE, 2014, pp. 522–529. doi:10.1109/BigData.Congress.2014.82.

[6] L. Bellatreche, S. Khouri and N. Berkani, Semantic data warehouse design: From ETL to deployment A La Carte, in: *International Conference on Database Systems for Advanced Applications*, Springer, 2013, pp. 64–83. doi:10.1007/978-3-642-37450-0_5.

[7] R. Berlanga, O. Romero, A. Simitsis, V. Nebot, T. Pedersen, A. Abelló Gamazo and M.J. Aramburu, *Semantic Web Technologies for Business Intelligence* (2011). doi:10.4018/978-1-61350-038-5.ch014.

[8] M.H. Bhuiyan, A. Bhattacharjee and R.P.D. Nath, DB2KB: A framework to publish a database as a knowledge base, in: *2017 20th International Conference of Computer and Information Technology (ICCIT)*, IEEE, 2017, pp. 1–7. doi:10.1109/ICCITECHN.2017.8281832.

[9] C. Bizer, T. Heath and T. Berners-Lee, Linked data: The story so far, in: *Semantic Services, Interoperability and Web Applications: Emerging Concepts, IGI Global*, 2011, pp. 205–227. doi:10.4018/978-1-60960-593-3.ch008.

[10] A. Bogner, B. Littig and W. Menz, *Das Experteninterview*, Springer, 2005. doi:10.1007/978-3-322-93270-9.

[11] M. Casters, R. Bouman and J. Van Dongen, *Pentaho Kettle Solutions: Building Open Source ETL Solutions with Pentaho Data Integration*, John Wiley & Sons, 2010.

[12] C. Ciferri, R. Ciferri, L. Gómez, M. Schneider, A. Vaisman and E. Zimányi, Cube algebra: A generic user-centric model and query language for OLAP cubes, *International Journal of Data Warehousing and Mining (IJDWM)* **9**(2) (2013), 39–65. doi:10.4018/jdwm.2013040103.

[13] D. Colazzo, F. Goasdoué, I. Manolescu and A. Roatiş, RDF analytics: Lenses over semantic graphs, in: *WWW*, ACM, 2014, pp. 467–478. doi:10.1145/2566486.2567982.

[14] P. Cudré-Mauroux, Leveraging Knowledge Graphs for Big Data Integration: The XI Pipeline, *Semantic Web* (2020), 1–5. doi:10.3233/SW-190371.

[15] R. Cyganiak, D. Reynolds and J. Tennison, The RDF Data Cube Vocabulary, W3C Recommendation, 2014, W3C, 2014, https://www.w3.org/TR/2012/WD-vocab-data-cube-20120405/.

[16] R.P.D. Deb Nath, Aspects of Semantic ETL, Aalborg Universitetsforlag, 2020.

[17] R.P.D. Deb Nath, K. Hose, T.B. Pedersen, O. Romero and A. Bhattacharjee, SETL$_{\mathrm{BI}}$: An integrated platform for semantic business intelligence, in: *Companion Proceedings of the Web Conference 2020*, 2020, pp. 167–171. doi:10.1145/3366424.3383533.

[18] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens and R. Van de Walle, RML: A generic language for integrated RDF mappings of heterogeneous data, 2014.

[19] L. Etcheverry, S.S. Gomez and A. Vaisman, Modeling and querying data cubes on the Semantic Web, 2015, preprint, available at: arXiv:1512.06080.

[20] L. Etcheverry, A. Vaisman and E. Zimányi, Modeling and querying data warehouses on the semantic web using QB4OLAP, in: *DaWak*, Springer, 2014, pp. 45–56. doi:10.1007/978-3-319-10160-6_5.

[21] L. Galárraga, K. Ahlstrøm, K. Hose and T.B. Pedersen, Answering provenance-aware queries on RDF data cubes under memory budgets, in: *International Semantic Web Conference*, Springer, 2018, pp. 547–565. doi:10.1007/978-3-030-00671-6_32.

[22] L. Galárraga, K.A.M. Mathiassen and K. Hose, QBOAirbase: The European air quality database as an RDF cube, in: *International Semantic Web Conference (Posters, Demos & Industry Tracks)*, 2017.

[23] N. Gür, J. Nielsen, K. Hose and T.B. Pedersen, GeoSemOLAP: Geospatial OLAP on the Semantic Web made easy, in: *Proceedings of the 26th International Conference on World Wide Web Companion*, ACM, 2017, pp. 213–217. doi:10.1145/3041021.3054731.

[24] N. Gür, T.B. Pedersen, E. Zimányi and K. Hose, A foundation for spatial data warehouses on the semantic web, *Semantic Web* **9**(5) (2018), 557–587. doi:10.3233/SW-170281.

[25] S. Harris, A. Seaborne and E. Prud'hommeaux, *SPARQL 1.1 Query Language, W3C Recommendation 21(10)*, 2013, https://www.w3.org/TR/sparql11-query/.

[26] A. Harth, K. Hose and R. Schenkel, *Linked Data Management*, CRC Press, 2014.

[27] T. Heath and C. Bizer, Linked data: Evolving the web into a global data space, *Synthesis Lectures on the Semantic Web: Theory and Technology* **1**(1) (2011), 1–136. doi:10.2200/S00334ED1V01Y201102WBE001.

[28] M. Hilal, C.G. Schuetz and M. Schrefl, An OLAP endpoint for RDF data analysis using analysis graphs, in: *ISWC*, 2017.

[29] D. Ibragimov, K. Hose, T.B. Pedersen and E. Zimányi, Towards exploratory OLAP over linked open data – a case study, in: *Enabling Real-Time Business Intelligence*, Springer, 2014, pp. 114–132. doi:10.1007/978-3-662-46839-5_8.

[30] D. Ibragimov, K. Hose, T.B. Pedersen and E. Zimányi, Processing aggregate queries in a federation of SPARQL endpoints, in: *European Semantic Web Conference*, Springer, 2015, pp. 269–285. doi:10.1007/978-3-319-18818-8_17.

[31] D. Ibragimov, K. Hose, T.B. Pedersen and E. Zimányi, Optimizing aggregate SPARQL queries using materialized RDF views, in: *International Semantic Web Conference*, Springer, 2016, pp. 341–359. doi:10.1007/978-3-319-46523-4_21.

[32] K.A. Jakobsen, A.B. Andersen, K. Hose and T.B. Pedersen, Optimizing RDF data cubes for efficient processing of analytical queries, in: *COLD*, 2015.

[33] P. Jovanovic, Ó. Romero Moral, A. Simitsis, A. Abelló Gamazo, H. Candón Arenas and S. Nadal Francesch, Quarry: Digging up the gems of your data treasury, in: *Proceedings of the 18th International Conference on Extending Database Technology*, 2015, pp. 549–552. doi:10.5441/002/edbt.2015.55.

[34] E. Kalampokis, B. Roberts, A. Karamanou, E. Tambouris and K.A. Tarabanis, Challenges on developing tools for exploiting linked open data cubes, in: *SemStats@ ISWC*, 2015. doi:10.1.1.703.6021.

[35] B. Kämpgen and A. Harth, No size fits all – running the star schema benchmark with SPARQL and RDF aggregate views, in: *Extended Semantic Web Conference*, Springer, 2013, pp. 290–304, doi:10.1007/978-3-642-38288-8_20.

[36] B. Kämpgen, S. O'Riain and A. Harth, Interacting with statistical linked data via OLAP operations, in: *Extended Semantic Web Conference*, Springer, 2012, pp. 87–101. doi:10.1007/978-3-662-46641-4_7.

[37] R. Kimball, *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*, John Wiley & Sons, Inc., 1996.

[38] T. Knap, P. Hanečák, J. Klímek, C. Mader, M. Nečaskỳ, B. Van Nuffelen and P. Škoda, UnifiedViews: An ETL tool for RDF data management, *Semantic Web* **9**(5) (2018), 661–676. doi:10.3233/SW-180291.

[39] E.V. Kostylev, J.L. Reutter and M. Ugarte, CONSTRUCT queries in SPARQL, in: *18th International Conference on Database Theory (ICDT 2015), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik*, 2015. doi:10.4230/LIPIcs.ICDT.2015.212.

[40] J. Li, J. Tang, Y. Li and Q. Luo, Rimom: A dynamic multistrategy ontology alignment framework, *IEEE Transactions on Knowledge and data Engineering* **21**(8) (2008), 1218–1232. doi:10.1109/TKDE.2008.202.

[41] Y. Marketakis, N. Minadakis, H. Kondylakis, K. Konsolaki, G. Samaritakis, M. Theodoridou, G. Flouris and M. Doerr, X3ML mapping framework for information integration in cultural heritage and beyond, *International Journal on Digital Libraries* **18**(4) (2017), 301–319. doi:10.1007/s00799-016-0179-1.

[42] J.S. Molléri, K. Petersen and E. Mendes, An empirically evaluated checklist for surveys in software engineering, *Information and Software Technology* **119** (2020), 106240. doi:10.1016/j.infsof.2019.106240.

[43] R.P.D. Nath, K. Hose and T.B. Pedersen, Towards a programmable semantic extract-transform-load framework for semantic data warehouses, in: *Acm Eighteenth International Workshop on Data Warehousing and Olap (dolap 2015)*, ACM, 2015. doi:10.1145/2811222.2811229.

[44] R.P.D. Nath, K. Hose, T.B. Pedersen and O. Romero, SETL: A programmable semantic extract-transform-load framework for semantic data warehouses, *Information Systems* **68** (2017), 17–43. doi:10.1016/j.is.2017.01.005.

[45] R.P.D. Nath, H. Seddiqui and M. Aono, Resolving scalability issue to ontology instance matching in semantic web, in: *2012 15th International Conference on Computer and Information Technology (ICCIT)*, IEEE, 2012, pp. 396–404. doi:10.1109/ICCITechn.2012.6509778.

[46] R.P.D. Nath, M.H. Seddiqui and M. Aono, An efficient and scalable approach for ontology instance matching, *JCP* **9**(8) (2014), 1755–1768. doi:10.4304/jcp.9.8.1755-1768.

[47] V. Nebot and R. Berlanga, Building data warehouses with semantic web data, *Decision Support Systems* **52**(4) (2012), 853–868. doi:10.1016/j.dss.2011.11.009.

[48] D. Pedersen, J. Pedersen and T.B. Pedersen, Integrating XML data in the TARGIT OLAP system, in: *Proceedings. 20th International Conference on Data Engineering*, IEEE, 2004, pp. 778–781. doi:10.1109/ICDE.2004.1320045.

[49] D. Pedersen, K. Riis and T.B. Pedersen, Query optimization for OLAP-XML federations, in: *Proceedings of the 5th ACM International Workshop on Data Warehousing and OLAP*, ACM, 2002, pp. 57–64. doi:10.1145/583890.583899.

[50] J. Pérez, M. Arenas and C. Gutierrez, Semantics and complexity of SPARQL, in: *International Semantic Web Conference*, Vol. 4273, Springer, 2006, pp. 30–43. doi:10.1007/11926078_3.

[51] I. Petrou, G. Papastefanatos and T. Dalamagas, 2013, pp. 1–3, Publishing census as linked open data: a case study. doi:10.1145/2500410.2500412.

[52] A. Polleres, A. Hogan, R. Delbru and J. Umbrich, RDFS and OWL reasoning for linked data, in: *Reasoning Web. Semantic Technologies for Intelligent Data Access*, Springer, 2013, pp. 91–149. doi:10.1007/978-3-642-39784-4_2.

[53] L. Richardson and S. Ruby, *RESTful Web Services*, "O'Reilly Media, Inc.", 2008.

[54] J. Rouces, G. De Melo and K. Hose, Heuristics for connecting heterogeneous knowledge via FrameBase, in: *European Semantic Web Conference*, Springer, 2016, pp. 20–35. doi:10.1007/978-3-319-34129-3_2.

[55] J. Rouces, G. De Melo and K. Hose, FrameBase: Enabling integration of heterogeneous knowledge, *Semantic Web* **8**(6) (2017), 817–850. doi:10.3233/SW-170279.

[56] M.H. Seddiqui, S. Das, I. Ahmed, R.P.D. Nath and M. Aono, Augmentation of ontology instance matching by automatic weight generation, in: *Information and Communication Technologies (WICT), 2011 World Congress on*, IEEE, 2011, pp. 1390–1395. doi:10.1109/WICT.2011.6141452.

[57] J.F. Sequeda and D.P. Miranker, Ultrawrap: SPARQL execution on relational data, *Journal of Web Semantics* **22** (2013), 19–39. doi:10.1016/j.websem.2013.08.002.

[58] D. Skoutas and A. Simitsis, Ontology-based conceptual design of ETL processes for both structured and semi-structured data, *IJSWIS* **3**(4) (2007), 1–24. doi:10.4018/jswis.2007100101.

[59] M. Thenmozhi and K. Vivekanandan, An ontological approach to handle multidimensional schema evolution for data warehouse, *International Journal of Database Management Systems* **6**(3) (2014), 33. doi:10.5121/ijdms.2014.6303.

[60] C. Thomsen and T. Bach Pedersen, pygrametl: A powerful programming framework for extract-transform-load programmers, in: *Proceedings of the ACM Twelfth International Workshop on Data Warehousing and OLAP*, 2009, pp. 49–56. doi:10.1145/1651291.1651301.

[61] A. Vaisman and E. Zimányi, *Data Warehouse Systems: Design and Implementation*, Springer, 2014.

[62] J. Varga, A.A. Vaisman, O. Romero, L. Etcheverry, T.B. Pedersen and C. Thomsen, Dimensional enrichment of statistical linked open data, *Journal of Web Semantics* **40** (2016), 22–51. doi:10.1016/j.websem.2016.07.003.

[63] X. Yin and T.B. Pedersen, Evaluating XML-extended OLAP queries based on physical algebra, *Journal of Database Management (JDM)* **17**(2) (2006), 85–116. doi:10.4018/jdm.2006040105.