SETL: A Programmable Semantic Extract-Transform-Load Framework for Semantic Data Warehouses

Rudra Pratap Deb Nath[a,b,*], Katja Hose[a], Torben Bach Pedersen[a], Oscar Romero[b]

[a] *Aalborg University, Aalborg, Denmark*
[b] *Universitat Politècnica de Catalunya, BarcelonaTech, Barcelona, Spain*

Abstract

In order to create better decisions for business analytics, organizations increasingly use external structured, semi-structured, and unstructured data in addition to the (mostly structured) internal data. Current Extract-Transform-Load (ETL) tools are not suitable for this "open world scenario" because they do not consider semantic issues in the integration processing. Current ETL tools neither support processing semantic data nor create a semantic Data Warehouse (DW), a repository of semantically integrated data.

This paper describes our programmable Semantic ETL (SETL) framework. SETL builds on Semantic Web (SW) standards and tools and supports developers by offering a number of powerful modules, classes, and methods for (dimensional and semantic) DW constructs and tasks. Thus it supports semantic data sources in addition to traditional data sources, semantic integration, and creating or publishing a semantic (multidimensional) DW in terms of a knowledge base. A comprehensive experimental evaluation comparing SETL to a solution made with traditional tools (requiring much more hand-coding) on a concrete use case, shows that SETL provides better programmer productivity, knowledge base quality, and performance.

Keywords: ETL, RDF, Semantic Integration, Data Warehouse,

*Corresponding author
Email address: rudra@cs.aau.dk (Rudra Pratap Deb Nath)

1. Introduction

Business Intelligence (BI) tools support intelligent business decisions by analyzing available organizational data. Data Warehouses (DWs) are used to store large data volumes from different operational databases in enterprises, and On-Line Analytical Processing (OLAP) queries are applied on DWs to answer business analytical questions. Extract-Transform-Load (ETL) is the backbone process of a DW, and the ETL design and deployment takes up to 80% of the time in DW projects [1]. To support data analyses or OLAP queries on it, the underlying schema of a DW is represented using the Multidimensional (MD) model. In the MD model, data are categorized as either facts with associated numerical measures or dimensions that characterize the facts [2]. This model represents any interesting observation of the domain (i.e., measures) in its context (i.e., dimensions) [3].

Nowadays, the Web is also an important source of information. Moreover, Semantic Web (SW) technologies and the Linked Data (LD) principles inspire organizations to publish and share their data using the Resource Description Framework (RDF) [4]. The role of data semantics in such sources is defined by different facets: using Internationalized Resource Identifiers (IRIs) to uniquely identify resources globally, providing common terminology, semantically linking published information, and providing further knowledge to allow reasoning [5]. The number of semantic data sources is ever increasing over time, and the growth rate of the data in the SW is faster than the computational power of computers [6]. As a result, besides analyzing internal data available in a DW, it is essential to incorporate external data from various (semantic) sources into the DW to derive the needed business knowledge. For example, companies want to include product reviews, customer complains, competitors' status from the Web in their analytical process besides the internal sales and customer data [3].

The inclusion of external data, especially RDF data, however, raises several

challenges for integration and transformation in comparison to the traditional ETL process. One of the drawbacks of using RDF data in the corporate analysis process is that data sometimes do not have any schema (e.g., only instances are delivered with implicit schema) [7], have a poor schema where not all the schema constructs are explicitly defined (e.g., only taxonomies are depicted), or complex schema (e.g., other more expressive ontological languages, such as OWL, are used). Unlike the relational data model, the RDF data model does not impose a common schema for all instances. In contrast, it provides flexible means to tackle different schema information as well as its evolution. This flexible nature of the RDF data model makes it suitable for representing and exchanging data in the SW. However, different sources may describe the same data in different ways, introducing semantic heterogeneity problems. Therefore, to build a successful DW system with heterogeneous data, the integration process should be able to deal with data semantics as a first-class citizen. Traditional ETL tools are unable to process such external data because they (1) do not support semantic data sources, i.e., they are not prepared to deal with semantic heterogeneity, (2) are entirely schema-dependent, and (3) do not focus on meaningful semantic relationships to integrate data from disparate sources [8]. Thus, a DW with both internal and external (semantic) data requires more powerful tools to define, enrich, integrate, and transform data semantically.

Until now, the DW community has timidly used SW technologies for considering the semantic issues in a DW [9], e.g., for data integration or describing the DW. SW technology aims at converting the 'Web of Documents' to the 'Web of Data' where data are presented and exchanged in a machine-readable and understandable format. In the SW, RDF is used for presenting and exchanging data in a machine-readable format. To express richer constraints on data, formal languages such as RDF Schema (RDFS) [10] and Web Ontology Language (OWL) [11] can be used in combination with the RDF data model to define Knowledge Bases (KBs). Although [3, 8, 12] have used SW technologies to design conceptual frameworks of different phases of ETL processes, no one has so far offered an integrated and implemented framework to build a Seman-

tic DW (SDW) that covers all of the phases of updating of sources, extraction, validation, and integration in one integrated platform.

This paper presents Semantic ETL (SETL), a unified framework for processing and integrating data semantically by bridging SW and DW technologies. SETL uses and extends SW tools and standards to overcome the limitations of the traditional ETL tools. Using SETL, the BI community can benefit by including semantic annotated data in their analytical processes and the SW community can benefit by having an MD view over semantic data for enabling OLAP-like analysis. Hence, it supports publishing better quality RDF datasets as well. The novel contributions of this paper are:

- We propose SETL, a unified framework for semantic ETL. The main tasks can be conducted within the framework are given below:
  (a) In addition to traditional relational data, SETL allows including semantically annotated data (RDF data) in the analytical process. To process a `Non Semantic Data Source`, it builds a semantic layer on top of the source.
  (b) To integrate data from disparate data sources, the user can define the intensional knowledge of SDW in the form of a terminology (TBox) [13] using the ontological constructs. In addition, it provides functionality to annotate the TBox with MD constructs, such as, dimensions, facts, levels, etc. Here, we use the QB4OLAP vocabulary [14] to define the MD constructs.
  (c) SETL provides functionality to produce semantic data (in RDF triples format) from the source data according to the (MD) semantics encoded in the TBox of the SDW.
  (d) SETL creates a SDW, a KB, composed of a TBox annotated with-/without MD constructs and an ABox (the instances of the TBox). It also provides functionality to semantically connect internal data with other internal/external data.
- We develop a high-level Python-based programmable framework that provides a number of powerful modules, classes, and methods for performing

the tasks mentioned above. It facilitate developers by providing a higher abstraction level that lowers the entry barriers. Thus, one of the contributions is to automatically map the high-level abstraction to executable code.

- Using SETL, we perform a comprehensive experimental evaluation by producing an MD SDW that integrates a `Semantic` and `Non Semantic Data Sources`. The evaluation shows that SETL improves considerably over the competing solutions/tools in terms of programmer productivity, KB quality, and performance.

This paper very significantly extends an earlier workshop paper [15] by (1) adding a sub-component *QB4OLAP* in the architecture of SETL that allows to define a target TBox with MD constructs, (2) adding a semantic layer on top of a `Non Semantic Data Source`, (3) extending the use case for creating an MD SDW which integrates a `Semantic Data Source` and a `Non Semantic Data Source`, (4) updating the semantic transformation algorithm to produce RDF triples according to MD constructs, (5) introducing a provenance graph for tracking how the IRIs used in the SDW are generated, and (6) comparing the experiment with other ETL tools/solutions.

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 details the source datasets and the target TBoxes of the SDW that we use as the running example. Section 4 gives an overview of SETL and its components. Section 5 describes the Definition Layer of SETL framework. The ETL Layer of the framework is described in Section 6. Section 7 describes the implementation of the framework in details. We evaluate SETL in terms of productivity, quality, and performance in Section 8. Section 9 describes related work. Finally, we conclude and give pointers to future work in Section 10.

2. Preliminary Definitions

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

RDF Graph. An RDF graph can be represented as a set of statements, called RDF triples. The three elements of a triple are subject, predicate, and object, respectively, and a triple represents a relationship between its subject and object by its predicate. Let I, B, and L be the sets of IRIs, blank nodes, and literals, respectively, where we denote the set of RDF terms $(I \cup B \cup L)$ as T and $(I \cap B \cap L) = \emptyset$. An IRI is an unique identifier that can be used to identify a resource globally (Web-scope). Blank nodes serve as locally-scoped identifier for resources that are unknown to the outside world. Literal are a set of lexical values enclosed with inverted commas. An RDF triple is defined as a 3-tuple (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 $G \subseteq (I \cup B) \times I \times T$ [16].

Knowledge Base. A Knowledge Base (KB) is typically composed of two Components: TBox and ABox. The TBox introduces the domain terminology. The ABox is the set of assertions representing individuals or instances. The ABox assertions must follow the TBox [13]. In this paper, we assume the components of a KB are described by a set of RDF triples, i.e., a KB is an RDF graph without distinguishing classes and instances.

The TBox is 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. A concept provides a general description of the features for similar types of resources. We use the terms "concept" and "class" interchangeably. Similar to other ontological formalisms, in this paper, we distinguish between object and datatype properties, depending on the RDF element used as object in the RDF triple. An object property relates concept instances,

represented as IRIs, while a datatype property is used to associate instances to literals. A^O describes a domain's concepts, properties, and the relationships among them. RDF-schema (RDFs) and the Web Ontology Language (OWL) provide basic constructs to define the TBox of a KB.

3. A Use case

This section describes the source datasets, and the TBoxes for integrating and understanding the knowledge of those datasets, used as the running example in this paper. We consider three Datasets: a Danish Agricultural dataset (DAD) [17], a Danish Business dataset (DBD) [18], and a European Union (EU) Farm Subsidy (`Subsidy`) dataset and integrate the datasets in two steps. First, we integrate the DAD with DBD to (re-)produce a SDW (called SETLKB) from the earlier paper [19]. Second, by integrating the `Subsidy` dataset and the produced SETLKB (which is considered as an external `Semantic Data Source`), we build the final MD SDW. In our context, the integration of different sources is achieved in an iterative and incremental manner. As proposed in [20], each iteration integrates a new source with the results of the previous integration (a single existing source table in the first iteration).

We build the SDW instead of a traditional DW because we want to publish the resulting DW on the Web in a format that other sources can easily integrate and make use of. SDW follows Linked Open Data (LOD) principles and provides IRIs that other people and sources can dereference and therefore receive information from them without having to issue an analytical query. Besides, it provides links to other Web accessible datasets. Therefore, in principle, we could even issue queries involving remote data at other sources, which is ~~also~~ something that a traditional DW cannot handle. Thus, our approach enables analyzing situational data (from external datasets) to personalize the user analysis, as discussed in [3, 20].

The DAD consists of three smaller Datasets: *Field*, *Organic Field*, and *Field Block*. All the datasets are available in Shape [21] format. The *Field* dataset

has 9 attributes and contains all registered fields in Denmark. This dataset overall contains information about 641,081 fields. The *Organic Field* dataset has 12 attributes and contains information about 52,060 organic fields. The *Field Block* dataset has 12 attributes for 314,648 field blocks [19].

The DBD is provided in CSV format. The dataset consists of two sub Datasets: *Company* and *Participant*. The *Company* dataset consists of 59 attributes and contains information about 603,667 companies and 659,639 production units. The *Participant* dataset describes the relations that exist between a participant and a legal unit [19]. Figure 1 shows how the datasets are connected to each other.

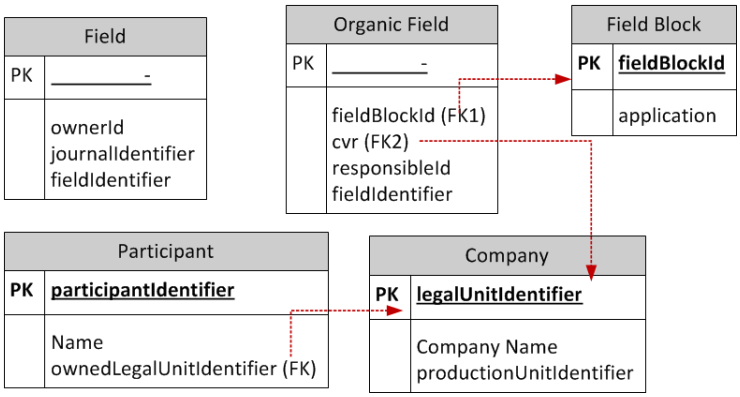


Figure 1: The schemas of the DAD and DBD datasets. *Field*, *Field Block* and *Organic Field* are spatially connected [19].

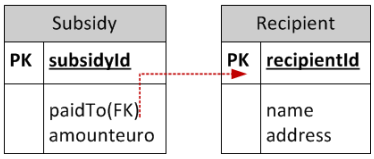


Figure 2: The conceptual schema of the Subsidy dataset. For simplicity, we do not show all columns.

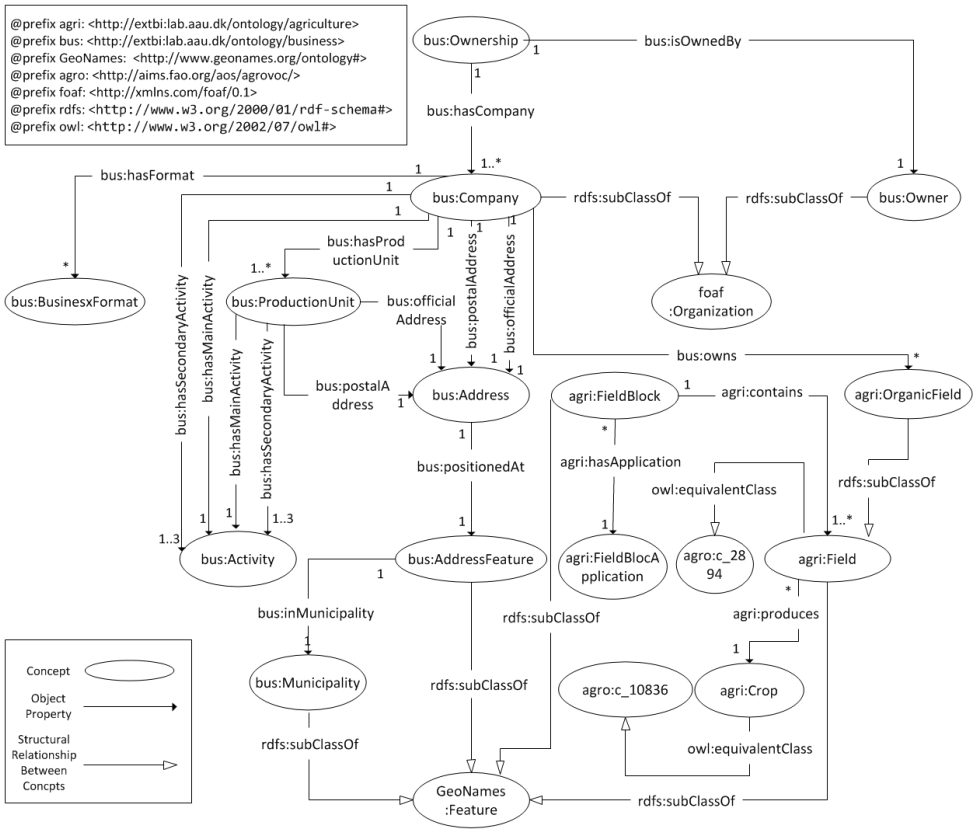


Figure 3: The TBox for the Danish Agricultural dataset and Danish Business dataset. The arrows show the cardinality of the relationship. Because of the large number of concept data type properties, they are not included in the figure.

Every year the EU provides subsidies to the farms of its member countries. We collect EU Farm subsidies for Denmark from `http://data.farmsubsidy.org/index.html?prefix=Raw/2014/`. The dataset contains two MS Access database Tables: *Recipient* and *Subsidy*. The *Recipient* table contains the information of 342,786 recipients who receive the subsidies and the *Subsidy* table contains the amount of subsidy given to the recipients in different years. In total, the *Subsidy* table contains 4,392,390 distinct records. The conceptual schema of the `Subsidy` dataset is shown in Figure 2.

To integrate the DAD and DBD datasets, we need to define a TBox of the SDW. The TBox of the SDW is illustrated in Figure 3. This is an up-

dated version of the TBox (ontology) described in [19]. We start the description from the `bus:Ownership` concept. This concept contains information about the owners of companies, the type of the ownership, and the start date of the company ownership. This concept is related to the `bus:Owner` concept through the `bus:isOwnedBy` property. The `bus:Owner` includes the details of the owners. The `bus:Ownership` has a one-to-many relationship with the `bus:Company` concept and they are related to each other through the property `bus:hasCompany`. The `bus:Company` concept is related to the concepts `bus:BusinessFormat` and `bus:ProductionUnit`. The concepts `bus:Company` and `bus:ProductionUnit` are also related to the `bus:Activity` concept through one main activity and/or through one to three secondary activities. Each company and each production unit have a postal address and an official address. Therefore, both concepts are related with the `bus:Address` concept. Each address has an address feature and it is contained within a particular municipality. The `bus:Company` concept is also related to the concept `agri:OrganicField` through the relation `bus:owns` as each company may contain one or more organic fields. Through this relation, the business and agricultural datasets are connected. The `agri:Field` concept defines the structure for all the registered agricultural fields of Denmark. Thus, `agri:OrganicField` is a subclass of `agri:Field`. The `agri:Field` is also equivalent to the UN definition of a European field. A field produces a crop, thus, `agri:Field` is connected to `agri:Crop`. A field is contained within a field block. Each field block has an application. The concepts `bus:AddressFeature`, `bus:Municipality`, `agri:Field`, `agri:FieldBlock` are defined as subclasses of the `GeoNames:Feature` concept in the GeoNames ontology. According to the semantics encoded in the TBox, we produce a SDW named SETLKB.

We create an MD SDW by integrating the produced SETLKB and the `Subsidy` dataset. The *Recipient* table in the `Subsidy` dataset contains the information of recipient id, name, address, and etc. From the SETLKB, we can extract information of the owner of a company who may receive the EU farm subsidy. The TBox of the SDW is shown in Figure 4, where the concept

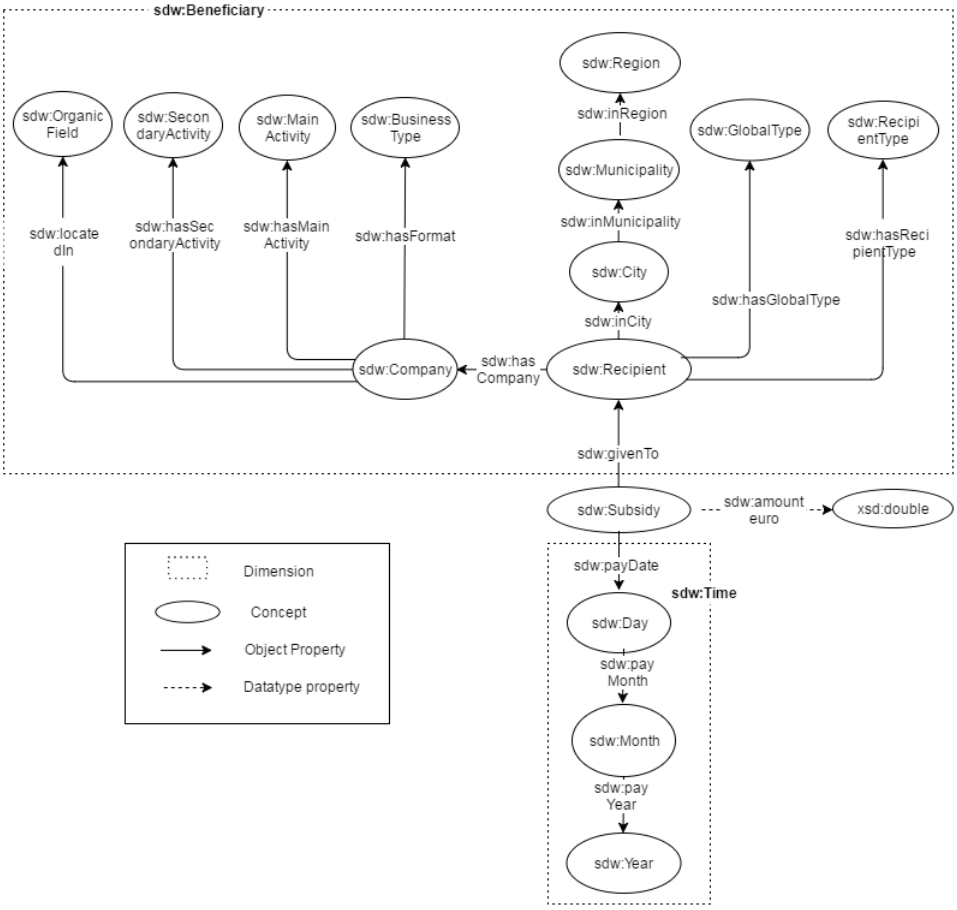


Figure 4: The TBox of our example SDW. Because of the large number, not all data properties of the dimensions/levels are shown.

`sdw:Subsidy` represents the factual concept[1] of the MD SDW and `sdw:amounteuro` is the measure. The SDW has two Dimensions: `sdw:Beneficiary` and `sdw:Time`. The dimensions are shown by a box with dotted line in Figure 4. Here, each level of the dimensions is represented by a concept and the hierarchies of the dimensions, i.e., how the levels of the dimensions are connected to each other through object properties, are also shown.

[1] A factual concept is the concept that provides a general description of the features for the facts of the DW.

4. SETL Framework Overview

In this section, we present the overview of our Semantic ETL framework (SETL in short). SETL follows a demand-driven approach to design a (MD) SDW. The first step of this approach is to identify and analyze the information requirements of business users and decision makers. Then, based on the gathered requirements, the next two steps of the DW are to build the DW schema and to build the ETL [22]. As the data in a SDW should be semantically connected, we assume the SDW to be a KB and it allows to produce an MD schema for the SDW to benefit from OLAP. The semantic ETL process populates the SDW from the data sources according to the semantics captured in the schema.

Our discussion focuses on SETL's architecture and its main components that support the different steps of creating a SDW. Requirement engineering (RE) is the process of identifying the needs of involved stakeholders and modeling and documenting those requirements in a form that is comprehensible, analyzable and communicable [23]. RE in the DW is itself a research topic and it is beyond the scope of this paper. We direct readers to [22, 23] for RE. The main steps that SETL supports are: defining a TBox for the SDW based on the domain of interest, extracting data from multiple heterogeneous data sources, transforming the source data into RDF triples according to the target TBox, linking the data internally and externally, and loading the data into a triple store, and/or publishing the data on the Web as Linked Data (LD).

Figure 5 illustrates how the components of SETL framework are connected to each other. We divide the framework into three Layers: Definition Layer, ETL Layer, and Data Warehouse Layer. The red-colored dashed lines in Figure 5 separate the layers. In the Definition Layer, the SDW schema, sources, and the mappings among the sources and the target are defined. The *SDW TBox Definition* component is used to define a TBox describing the relevant data and the SDW schema based on the requirements. The sub-component *QB4OLAP* is used to define the MD semantics of the SDW. Using the *Define Mapping* component, the user can define the mappings between a source TBox and the

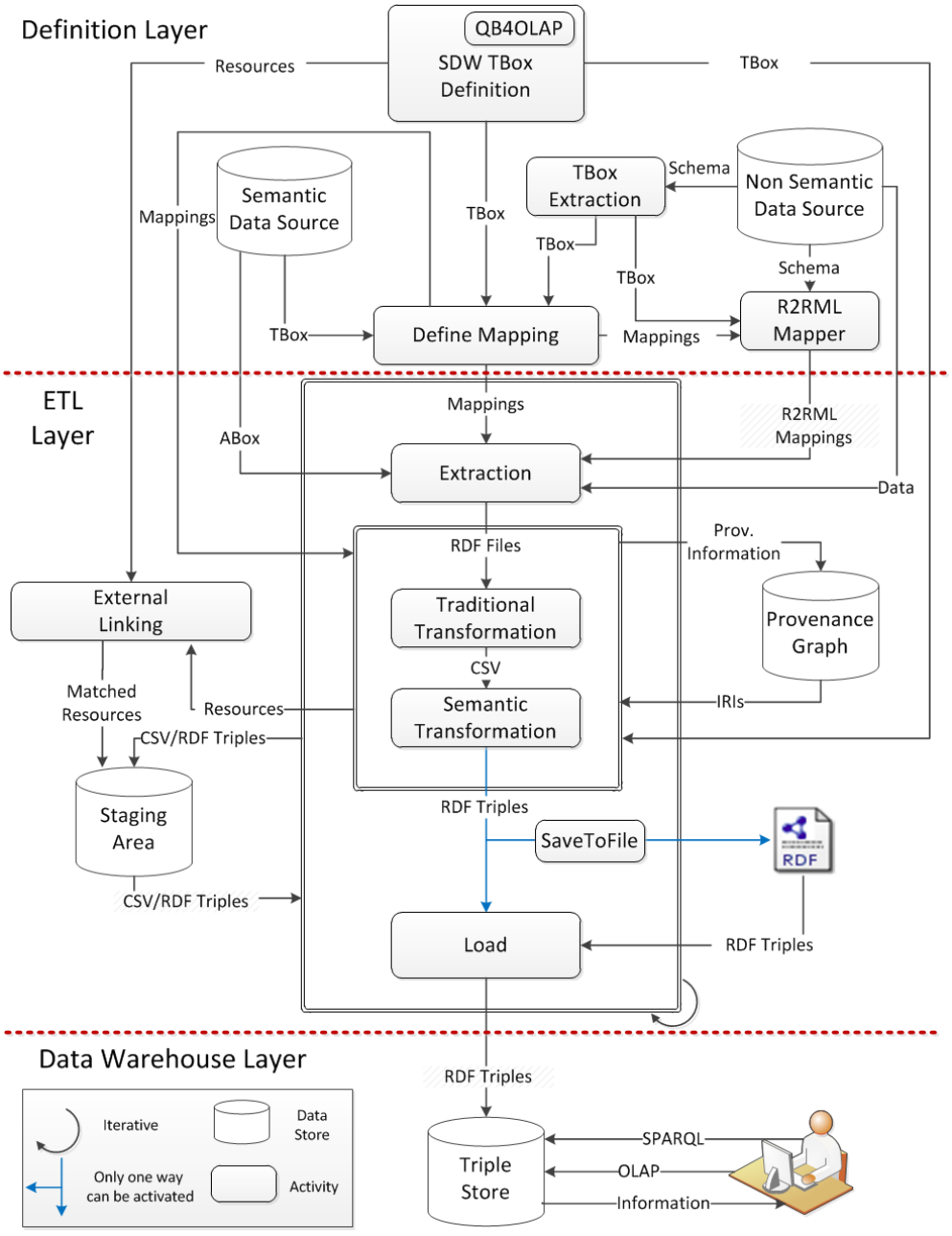


Figure 5: SETL architecture.

target TBox. To map between a `Non Semantic Data Source` and the target TBox, the *TBox Extraction* component generates a TBox representing the `Non Semantic Data Source`. In the ETL Layer, an ETL process to populate the SDW from sources are designed. The *Extraction* component extracts data from multiple sources, the *Traditional Transformation* component cleanses and formats (e.g., unique value generation, null value handling, noisy data filtering etc.) the extracted data and stores them in a `Staging Area`. The `Staging Area` is a storage used to keep the intermediate results of the ETL sub-processes. Then, the *Semantic Transformation* converts the data into RDF triples according to the target TBox. As a sub-task *Semantic Transformation* stores the meta information of concepts, properties, instances, and IRIs used in the SDW into the `Provenance Graph`. The *External Linking* component links the resources in the created RDF dataset to other external resources. The *SaveToFile* component writes the created RDF dataset into a file on disk. Finally, the *Load* component can either directly load the RDF triples created by the *Semantic Transformation* component or load from the RDF dump file into the `triple store`, which can directly be queried by the user. Extraction-Transformation-Load is an iterative block repeated for each ETL flow (shown as a curved arrow in Figure 5). The Data Warehouse Layer concerns where the transformed semantic data should be stored. SPARQL queries can be used to analyze the stored data. The SDW defined in MD fashion can also be integrated with an OLAP tools to perform OLAP queries. Here, we focus on building an ETL process. The following sections describe the Definition Layer and ETL Layer along with their components in more details.

5. Definition Layer

In this section, we describe the Definition Layer of SETL. This layer integrates different components required to define the schema of a SDW, to define the different sources that feed data in the SDW, and to define the mappings between the sources and the target. The following sections describe the different

components of the Definition Layer.

5.1. SDW TBox Definition

The data in a SDW are semantically connected with other internal and/or external data. Therefore, capturing the semantics of the data at the conceptual level is indispensable. SETL uses ontological constructs to design a SDW (in our context the SDW TBox) because of the following reasons. First, ontologies allow for a semantic integration of disparate data sources as we can explicitly define how two concepts of an ontology are structurally related and how they are associated with different properties. Second, given the cubes share dimensions, it facilitates drill-across operations [24] by basic graph operations during the integration of cubes. Third, it is machine-readable and allows us to automate the ETL modelling tasks. Fourth, compared to other representations, it is easier to evolve the DW. Fifth, it allows to preserve the semantics of the `Semantic Data Sources`.

As mentioned in Section 2, some standard languages, such as RDFS or OWL can be used to describe a TBox. They both provide basic constructs to define the formal semantics of the TBox. OWL uses `owl:Class` and `rdfs:Property` to define concepts and properties of the TBox, uses `rdfs:subClassOf` and `rdfs:subPropertyOf` to define hierarchical relationships among concepts and among properties, respectively, and uses `rdfs:domain` and `rdfs:range` to associate properties with concepts.

On-Line Analytical Processing (OLAP) is a technology to analyze the data available in a DW to support decision making [25]. As OLAP is on-line, it should provide answers quickly. To enable OLAP queries, the DW is represented using the MD model. The central attraction of the MD model of a business is its simplicity and the easy and intuitive way of making analytical queries [26]. 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). A fact is the interesting thing or process to be analyzed (for example, analysis of subsidy given by the EU) and the attributes of the fact are

called measures (e.g., amount of subsidy), usually represented as numeric values. A dimensions is organized into hierarchies, composed of a number of levels, which permit users to explore and aggregate measures at various levels of detail. For example, the *Address* hierarchy (*recipient* $\rightarrow$ *Municipality* $\rightarrow$ *Region* $\rightarrow$ *Country*) of the *Beneficiary* dimension allows to aggregate the subsidy amount at various levels of detail. Therefore, to enable OLAP, we support the MD representation of the SDW.

Although the version of the framework described in the workshop paper [15] allows to define a TBox using OWL, it does not support to define MD constructs. We extend this version by adding the *QB4OLAP* sub-component to support MD constructs. To describe the MD semantics at the TBox level, we use the QB4OLAP vocabulary [27]. QB4OLAP is used to annotate the TBox with MD constructs and is based on the RDF Data Cube (QB) which is the W3C standard to publish MD data on the Web [28]. The QB is mostly used for analyzing statistical data and does not adequately support OLAP MD constructs. Therefore, we direct to QB4OLAP. Figure 6 depicts the QB4OLAP vocabulary [27]. 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 RDF classes, and non-capitalized terms represent RDF properties. Capitalized terms in italics represent classes with no instances. An arrow with black triangle head from class A to class B, labeled *pro* means that *pro* is an RDF property with domain A and range B. White triangles represent sub-classes or sub-properties.

In QB4OLAP, the concept `qb:DataStructureDefinition` is used to define the structure of a cube in terms of dimensions, measures, and attributes. To define dimensions, levels, level-attributes and hierarchies, the concepts `qb4o:DimensionProperty`, `qb4o:LevelProperty`, `qb4o:LevelAttribute`, and `qb4o:Hierarchy` are used, respectively. The association between a level-attribute and a level is defined by `qb4o:hasAttribute` property. The hierarchies are connected with dimensions via the property `qb4o:hasHierarchy`. Hierarchies are composed of pairs of levels, which are defined by the concept `qb4o:HierarchyStep`.

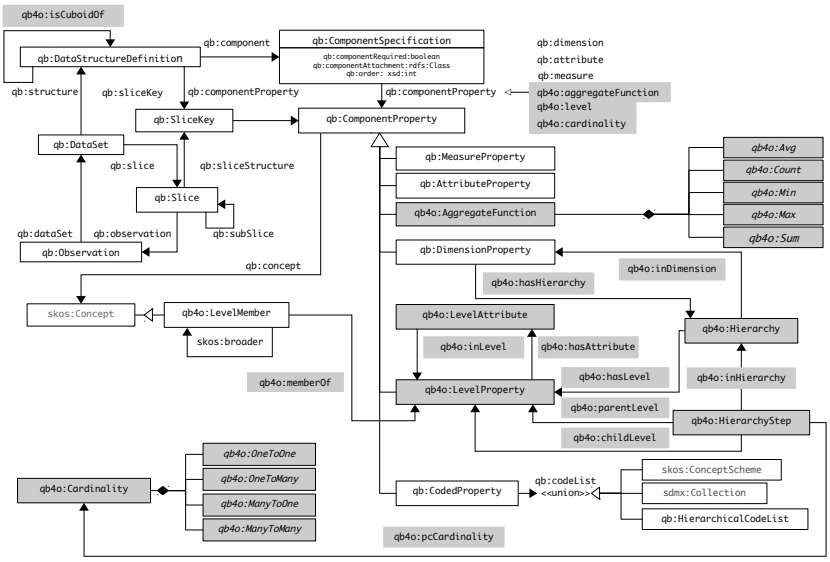
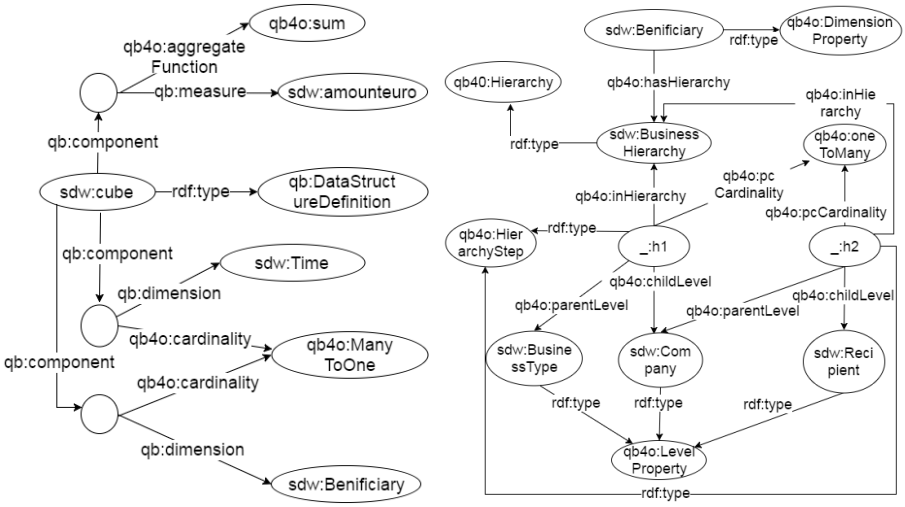


Figure 6: QB4OLAP vocabulary.

The reason of defining the hierarchy as a collection of pairs is to establish a rollup relationship between the levels of a pair. A rollup relation is defined by `qb4o:RollupProperty` and the cardinality of the rollup relation is stated using the `qb4o:pcCardinality`. Each pair of levels is connected to a rollup relation via `qb4o:rollup`. The role of the levels of a pair is distinguished using the property `qb4o:parentLevel` and `qb4o:childLevel`. Pairs are connected with hierarchies through the property `qb4o:inHierarchy` [27]. Therefore, the set of concepts C and the set of properties P in the TBox contains $C_m \subset C$ and $P_m \subset P$ created from the QB4OLAP constructs, where

$C_m = \{\texttt{qb:DataStructureDefinition}, \texttt{qb4o:DimensionProperty}, ..\}$
and $P_m = \{\texttt{qb4o:hasAttribute}, \texttt{qb4o:hasHierarchy}, \texttt{qb4o:inHierarchy}, ..\}$.

Figure 7a shows a part of the QB4OLAP cube structure for our use case. A cube may have several dimensions and measures. The cube (`sdw:cube`) has a set of components (blank nodes), which represent different measures, dimen-



(a) A fragment of the cube structure. **(b)** Beneficiary dimension definition.

Figure 7: Description of a part of the cube structure and a dimension of our running example using QB4OLAP.

sions, attributes, levels, and so on. A measure has a property to store numerical values in the dataset (e.g., `sdw:amount`) and an aggregation function (e.g., `qb4o:sum`). A measure can be analyzed according to different dimensions, e.g., `sdw:Beneficiary`, `sdw:Time`. Figure 7b illustrates a segment of `sdw:Benificiary` dimension. It has a hierarchy named `sdw:BusinessHierarchy`, which is composed of two hierarchy steps, namely `_:h1` and `_:h2`. Each hierarchy-step maintains the roles between the levels it contains, e.g., `_:h1` contains two levels, namely, `sdw:BusinessType`, and `sdw:Company`, and the parent and the child of `_:h1` are `sdw:BusinessType`, and `sdw:Company`, respectively. The cardinality of the step is defined by `qb4o:pcCardinality`.

SETL allows users to define a TBox manually. It allows to define various ontological constructs, such as, concepts, properties, and blank nodes individually as well as to capture how they relate to each other. The user first defines concepts and properties individually and then explicitly connects particular concepts to a set of properties. The MD constructs, such as, dimensions, levels, factual concepts, hierarchies, cube structures, cubes are represented by anno-

tating the concepts of the TBox using the QB4OLAP vocabulary. In order to denote a particular MD construct, the concept is annotated as a member of corresponding QB4OLAP class. For example, the triples (`sdw:Beneficiary rdf:type owl:Class`, `qb4o:DimensionProperty`.) represent that the concept `sdw:Beneficiary` is annotated as a dimension of the target TBox. At first, the user defines the concepts for dimensions, levels, level attributes, measures, and hierarchies, and then based on the defined concepts, the structure of the data cube, dataset and factual concept are defined. Internally, SETL indexes the set of properties connected to a particular concept. As user can also define a TBox using other ontology editors, such as Protege, SETL also allows to parse a given TBox.

5.2. Data Source

A data source is the source of data that can be used to populate a DW. We define a data source as a 2-tuple:

$$D = (ds, type)$$

where ds is the formal definition of the data source and $type$ is the type of the data source. The types of a data source can be either a `Semantic Data Source` (ss) or a `Non Semantic Data Source` (ns).

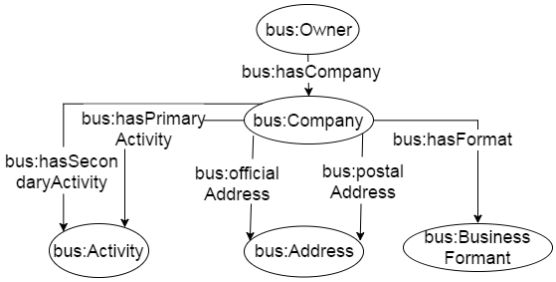


Figure 8: A segment of SETLKB TBox.

Semantic Data Source. A `Semantic Data Source` is a typical KB, which is discussed in Section 2. Therefore, a `Semantic Data Source` is defined as $D = (KB, ss)$. Figure 8 depicts a part of TBox shown in Figure 3. The semantic

graph to logically define a part of TBox and ABox of Figure 8 is depicted in Figure 9. In Figure 9, the segment of the graph above the vertical dashed line represents the TBox and below the dashed line represents sample instances (i.e., ABox). The semantic graph (RDF graph) is considered as a directed label graphs which conceptualize an RDF dataset. In the graph, subjects and objects of RDF triples are drawn by the labeled vertices and predicates are shown as directed labeled edges. To differentiate between the TBox and ABox, we draw the instances in ABox with rectangle, and literal-vertices with simple text.

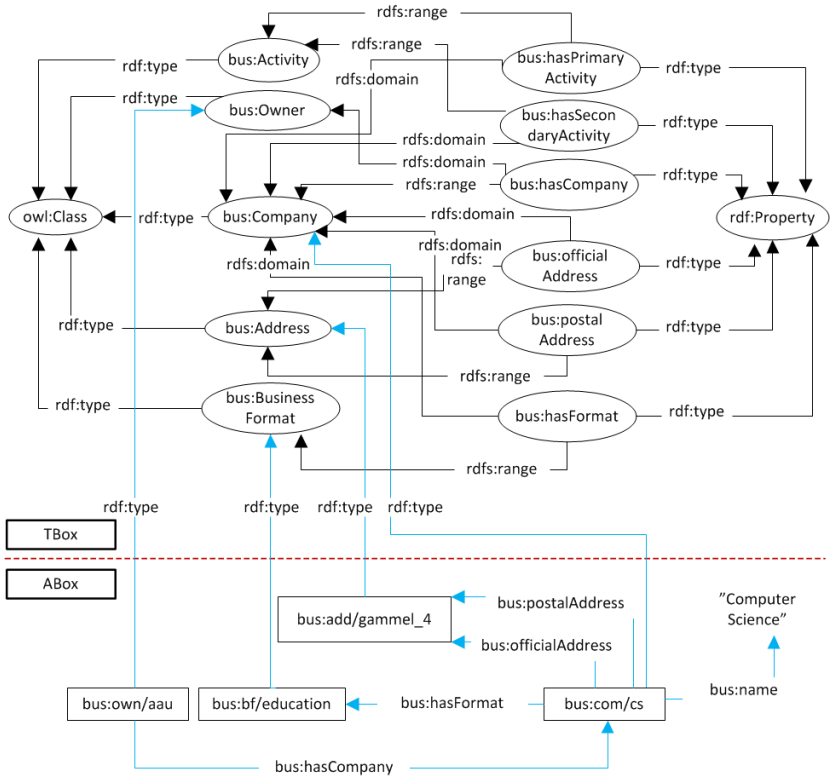


Figure 9: Semantic graph to represent a portion of TBox and ABox of Figure 8.

Typically, an RDF dataset is physically stored as an RDF dump file or in a triple store, for instance, Jena TDB, Virtuoso, Sesame store [29].

Non Semantic Data Source. A `Non Semantic Data Source` can be a relational database (RD), a shapefile, an XML file, an object-oriented database,

or a CSV. However, the current version of SETL supports RD, shapefile, and CSV, and the framework is designed in a way that allows it to be easily extended to support other formats as well.

For an RD, $D = (\mathcal{R}, ns)$, and $\mathcal{R}$ can be further defined as $\mathcal{R} = (\mathcal{S}, \mathcal{D})$, where $\mathcal{S}$ and $\mathcal{D}$ represent the schema and data of $\mathcal{R}$. To give the formal definition of a RD, we use the notation from [30]. An RD is composed of a set of tables $\mathcal{T}$. Each $t \in \mathcal{T}$ can be defined by 2-tuples: $t = ((C_t, PK_t, FK_t), R_t)$, where

$-(C_t, PK_t, FK_t)$ denotes the schema of the table t

$-C_t$ is the set of all columns of t

$-PK_t$ is the set of all primary key columns of t

$-FK_t$ is the set of foreign keys

$-R_t$ is the set of rows in t , which represents the data of table t .

Each foreign key $FK_{t,t'} \in FK_t$ is a subset of one or more columns and holds $\forall FK_{t,t'} \in FK_t, \forall r \in R_t : \exists t' \in \mathcal{T}, \exists r' \in R_{t'}\ where\ r(FK_{t,t'} = r'(PK_{t'}))$, i.e., each foreign key of a table must be a primary key of same/other table.

Table 1: The sample data of `Subsidy` dataset

(a) Subsidy

subsidyId	**paidTo**	**amount**
10611390	366894	617308
10611402	362146	6310

(b) Recipient

recipientId	**name**	**address**
362146	Jan'S Værksted	Gammel 4
366894	Videncentret For Landbrug	Agro Park 15

Therefore, considering all tables $\mathcal{T}$ in the RD, D becomes

$$D = ((\bigcup_{t \in \mathcal{T}}(C_t, PK_t, FK_t)), (\bigcup_{t \in \mathcal{T}}(R_t))), ns).$$

Figure 2 shows the conceptual schema of the `Subsidy` dataset. Table 1 shows some sample data of *Subsidy* and *Recipient* tables. We can define the `Subsidy` database as

`Subsidy` = (({({{`subsidyId`, `paidTo`, `amount`}, {`subsidyId`}, {`paidTo`}),
({`recipientId`, `name`, `address`}, {`recipientId`}, {})}), ({(“10611390”, “366894”,

"617308"), ("10611402", "362146", "6310")}, {("362146", "Jan'SVærksted", "Gammel4"), ("366894", "VidencentretForLandbrug", "AgroPark15")}), ns).

Typically, the tables of a RD are stored in a Relational Database Management System (RDBMS), for example, PostgreSQL, Oracle RDBMS, Microsoft SQL server etc.

A shapefile [21] is a file format that stores geometric location and attribute information of geographic features in vector format. Points, lines, and polygons can be used to describe a geographic feature. It can also be represented in the format of a table. This table is just like any other relational table except it contains a special geometry column. This column stores the actual geometry shape of geographic data. These are the edges and vertices locations that make up the spatial data. Typically, all the other columns in this table are the attributes information associated with the spatial data.

A CSV file is a comma separated values file, which allows data to be saved in a table structured format. Each line of the file is a row of the table and each row consists of one or more fields, separated by commas. Thus, shape and CSV files can be formalized using the notation of RD.

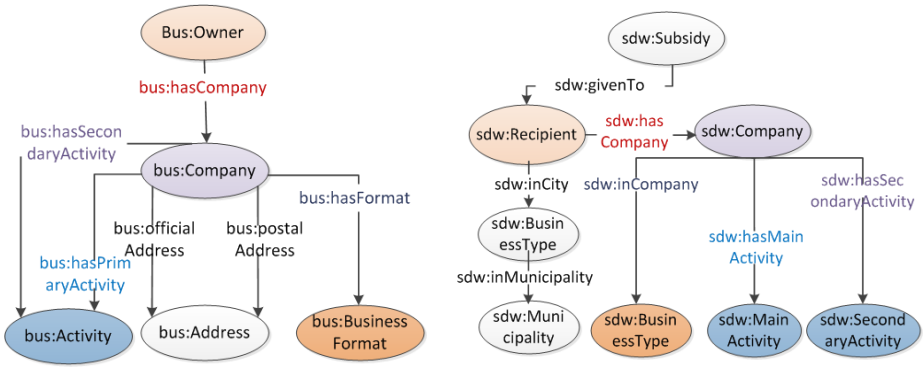
5.3. Define Mapping

A relationship between semantically similar but autonomously designed data needs to be established [31]. As data sources are highly heterogeneous in syntax, mapping should be done between sources and the target at the schema level. Mappings define the relationship between the elements (either concepts or properties) of a source and the corresponding elements in the target TBox [32].

Given two TBoxes T_S and T_T, a mapping maps an element in T_S to the corresponding element in T_T. T_S and T_T are called the source TBox and target TBox, respectively. We formally define an *TBox Mapping* as

$$Map(T_S, T_T) = \{(e_{s1}, e_{t1}, type_i) | e_{s1} \in T_S, e_{t1} \in T_T, type_i \in \{\texttt{skos:exact}, \texttt{skos:narrower}, \texttt{skos:broader}, \texttt{owl:sameAs}, \texttt{rdfs:subClassOf}, \texttt{rdfs:subPropertyOf}\}\}.$$

Each 3-tuple $(e_{s1}, e_{t1}, type_i)$ in $Map(T_S, T_T)$ represents that e_{s1} in T_S is mapped to e_{t1} in T_T with the relationship $type_i$. The relationship can be either *equivalence* relationship ($e_{s1} \equiv e_{t1}$) or *subsumption* relationship ($e_{s1} \sqsubseteq e_{t1}$) [33]. Different knowledge representation languages use different properties to represent *equivalence* and *subsumption* relationships. Therefore, the set of properties supporting the (*equivalence* and *subsumption*) relationships can be extended, but in the current version, we use, `skos` : `exact`, and `owl` : `sameAs` to represent equivalence relationship and `skos` : `narrower`, `skos` : `broader`, `rdfs` : `subClassOf`, and `rdfs` : `subPropertyOf` are used to represent *subsumption* relationship.



(a) T_S: Source TBox. **(b)** T_T: Target TBox.

Figure 10: Fragments of a source and the target TBoxes to be mapped. Same colors across the TBoxes indicate mapped elements.

Figure 10 shows the fragments of a source TBox (Figure 10a) and the target TBox (Figure 10b). The mappings from the source TBox to the target TBox are shown in Table 2. SETL allows users to define the mappings manually to keep them up to date with the needs of end-users.

Mapping for Semantic Data Source. As the schema of a `Semantic Data Source` is defined using a TBox and we assume that the TBox is integrated in the source, no additional step is required to define the mappings between the source and target TBoxes. However, as discussed in Section 1, sometimes RDF datasets do not include an explicit schema. In that case, a TBox should be derived from the RDF [7]. Currently, SETL does not provides this facility. It

Table 2: Mapping example from the source TBox to the target TBox

Entity in T_S	Mapped entity in T_T	Relation
bus:Owner	sdw:Recipient	*subsumption* (bus:Owner $\sqsupseteq$ sdw:Recipient)
bus:Activity	sdw:MainActivity	*subsumption* (bus:Activity $\sqsupseteq$ sdw:MainActivity)
bus:Activity	sdw:SecondaryActivity	*subsumption* (bus:Activity $\sqsupseteq$ sdw:SecondaryActivity)
bus:hasCompany	sdw:hasCompany	*equivalence*
bus:hasPrimaryActivity	sdw:hasPrimaryActivity	*equivalence*
bus:hasSecondaryActivity	sdw:hasSecondaryActivity	*equivalence*
bus:BusinessFormat	sdw:BusinessType	*equivalence*
bus:hasFormat	sdw:inCompany	*equivalence*

will be addressed in future.

Mapping for Non Semantic Data Source. In Figure 5, it is shown that the *Define Mapping* component takes a source and the target TBoxes as input and outputs the mapped elements across the TBoxes. Thus, a further step is required to extract the TBox from a `Non Semantic Data Source`. The following paragraph defines how a TBox is defined as a semantic layer on top of the underlying `Non Semantic Data Source`.

TBox Extraction TBox Extraction is the process of constructing a TBox (semi-) automatically from a given data source [34]. We define the extraction process f from a data source D by:

$$f : D \rightarrow \mathcal{TB}$$

where $\mathcal{TB}$ is the TBox derived from D. Given the heterogeneous types of sources (e.g., relational, xml, object-oriented) from where the ontologies to be derived, an instance of f should be defined for each type of sources [30].

Using the notation of a RD described in Section 5.2, we define the TBox extraction function for a RD as $f_{rd} : \mathcal{S} \rightarrow \mathcal{TB}$, where S is the schema of the given RD, composed of set of tables $\mathcal{T}$. For this function, we also use the notation and process from [30]. Table 3 shows the mapping between the elements of a RD schema and corresponding OWL constructs. Each table $t \in \mathcal{T}$ of the RD is mapped to an OWL class, each column $c \in C_t$ is mapped to a datatype property, and each foreign key $FK_{t,t'} \in C_t$ is mapped to an object property. The additional properties (e.g., `rdfs:domain`, `rdfs:range`) related to the main

Table 3: TBox extraction function from RD

x	$f_{rd}(x)$
$\forall t \in \mathcal{T}$	`owl:class`
$\forall c \in C_t$	`owl:DatatypeProperty` [`rdfs:domain` $= f_{rd}(t)$, `rdfs:range`=type(c)]
$\forall FK_{t,t'} \in C_t$	`owl:ObjectProperty` [`rdfs:domain` $= f_{rd}(t)$, `rdfs:range`$=f_{rd}(t')$]

property are shown within a square bracket in Table 3. For example, for an OWL property, the domain and range are also shown in Table 3. Figure 11 shows the extracted TBox of the `Subsidy` database shown in Figure 2.

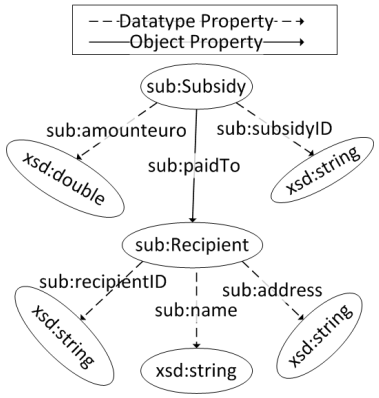


Figure 11: The extracted `Subsidy` TBox.

To map the relational data to RDF data using the extracted TBox the *R2RML Mapper* component is used. This component uses the R2RML mapping language, which is a W3C standard to define customized mappings from relational data to RDF [35]. This mapping is done by users. As all relational data may not be relevant to the target, we also consider the output of *Define Mapping* component in the *R2RML Mapper* (shown in Figure 5) to map only the relevant data. An R2RML mapping is itself represented as an RDF

graph. Listing 1 shows a segment of the R2RML mapping between the `Subsidy` dataset (in Table 1) and the `Subsidy` TBox (shown in Figure 11). An R2RML mapping document consists of one or more structures called TripleMaps (e.g., `sub:subsidyMap` and `sub:recipentMap` in Listing 1). The `rr:TriplesMap` class consists of three properties, namely, `rr:logicalTable`, `rr:subjectMap`, and `rr:predicateObjectMap`. Each TripleMap contains a reference to a logical table (e.g., Subsidy, Recipient) using the property `rr:logicalTable` (lines 5-6). A logical table can be a table, or a view or a SQL query. The property `rr:subjectMap` specifies the target class of TBox and the IRI generation process for each row (lines 7-10). The RDF triples generated from a row share the same subject. The property `rr:predicateObjectMap` defines a target property and the generation of the value of the property using `rr:objectMap`. The type of the RDF terms is either a constant, or a template, or a column value. A constant-valued term map always generates the same RDF terms. PredicateMaps are usually constant-valued (line 12). If the term map is a column-valued, the values in the specified column of the logical table will be used to generate the terms (lines 13-14). If it is template-valued, the terms will be generated based on the given template. A template is a string that concatenates several columns names or the base IRI and the values of one or more column names to generate unique IRIs. Line (8-9) creates IRIs for the subject of the triple using `rr:template` term map. The term map also have specified term type (IRI, Blank node, or Literal) [36].

Listing 1:**R2ML mapping from Subsidy TBox to Subsidy relational data**

```
@prefix rr: <http://www.w3.org/ns/r2rml\#>.
@prefix sub: <$http://extbi.lab.aau.dk/ontology/sub/> .
sub:subsidyMap
        a rr:TriplesMap ;
        rr:logicalTable [
        rr:tableName 'Subsidy' ] ;
        rr:subjectMap [
```

```
                rr:template 'http://extbi.lab.aau.dk/
                ontology/sub/subsidy/{subsidyId}' ;
                rr:class sub:Subsidy ; ] ;
        rr:predicateObjectMap [
                rr:predicateMap [rr:constant sub:subsidId
                ];
                rr:objectMap [
                        rr:column 'subsidyId' ;
                        rr:termType rr:Literal ;] ;
        ];
        rr:predicateObjectMap [
                rr:predicateMap sub:paidTo ;
                rr:objectMap [
                        rr:template 'http://extbi.lab.aau.
                dk/ontology/sub/recipient/{paidTo}' ; ] ;
        ];
        rr:predicateObjectMap [
                rr:predicateMap sub:amounteuro ;
                rr:objectMap [
                        rr:template "{amounteuro}.00" ;
                        rr:termType Literal ;] ;
        ];
sub:recipientMap
        a rr:TriplesMap ;
        ....
```

6. ETL Layer

In this section, we describe the ETL Layer of SETL. This layer integrates different components to design an ETL process to populate the SDW from

different data sources. The following sections describe the different components in details.

6.1. Extraction

Extraction is the process of acquiring data from sources. In this section we describe how to extract data from `Semantic Data Sources` and `Non Semantic Data Sources`.

Extraction from Semantic Data Source. SPARQL is the standard pattern matching language for querying a `Semantic Data Source`. It uses SELECT query to retrieve the desired output from the source. The output of the SPARQL SELECT query is a bag of bindings for the variables in the SPARQL query pattern. Unlike SELECT query, the SPARQL CONSTRUCT query is used to construct the triples from the `Semantic Data Source`, i,e., the output of the query itself is an RDF graph. As we need to retrieve the triples from a `Semantic Data Source`, in this section, we discuss the semantics of the basic SPARQL CONSTRUCT query.

To define the basic semantics of SPARQL CONSTRUCT query, we need to introduce the notion of triple patterns and basic graph pattern (BGP). A triple pattern is an RDF triple which allows query variables in any position of the triple, i.e., $t_p \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$, where V is a set of query variables that range over all RDF terms T, and $V \cap T = \emptyset$. A query variable $v \in V$ is led by the symbol $'?'$, e.g., $(?s\ p\ o)$. A BGP is a set of triple patterns connected via logical conjunctions. The SPARQL graph pattern expression is defined based on BGP.

The execution of triple patterns against a `Semantic Data Source` produces a set of solution mappings. A solution mapping μ is a partial function that maps V to T, $\mu : V \rightarrow T$. The domain of μ, denoted by $dom(\mu)$, is the subset of V for which μ is defined.

Let $\mathcal{K}$ and t_p be a `Semantic Data Source` and a triple pattern, and $var(t_p)$ denotes the set of query variables t_p contains. The evaluation of t_p against $\mathcal{K}$ is

the set of all mappings that can map t_p to a triple contained in $\mathcal{K}$, i.e.,

$$[\![t_p]\!]_{\mathcal{K}} = \{\mu \mid dom(\mu) = var(t_p) \; and \; \mu(t_p) \in \mathcal{K}\}.$$

where $\mu(t_p)$ is the triple obtained by replacing the variables in t_p according to μ. The SPARQL CONSTRUCT query makes the class of queries whose inputs and answers are RDF graphs [37]. Therefore, we define the answer $\texttt{ans}(t_p, \mathcal{K})$ of the SPARQL CONSTRUCT query as a set of triples matched t_p in $\mathcal{K}$, i.e.,

$$\texttt{ans}(t_p, \mathcal{K}) = \{\mu(t_p) \mid \mu \in [\![t_p]\!]_{\mathcal{K}} \; and \; \mu(t_p) \in \mathcal{K}\}.$$

If $B = \{t_{p1}, t_{p2}, t_{p3}, ...\}$ is a BGP, then the evaluation of B over $\mathcal{K}$ is

$$[\![B]\!]_{\mathcal{K}} = \{\mu \mid dom(\mu) = var(B) \; and \; \mu(B) \subseteq \mathcal{K}\}.$$

Here, $\mu(B)$ is the set of triples obtained by replacing the variables in the triple patterns of B according to μ. Therfore, the answer $\texttt{ans}(B, \mathcal{K})$ of the query is

$$\texttt{ans}(B, \mathcal{K}) = \mu(B) = \bigcup_{t_{pi} \in B} \{\mu(t_{pi}) | \mu \in [\![B]\!]_{\mathcal{K}} \; and \; \mu(t_{pi}) \in \mathcal{K})\}.$$

Consider the `Semantic Data Source` shown in Figure 9 and we want to construct the triples for the properties of `bus:Company`. Here, $t_{cp} =$ $(?property$ `rdfs` : `domain` `bus` : `Company`$)$, $dom(\mu) = ?property$. The evaluation of t_{cp} over DAB is

$$[\![t_{cp}]\!]_{DAB} = \{\{?property \rightarrow \texttt{bus} : \texttt{hasFormat}\}, \{?property \rightarrow \texttt{bus} : \texttt{officialAddress}\},$$
$$\{?property \rightarrow \texttt{bus} : \texttt{postalAddress}\}, \{?property \rightarrow \texttt{bus} : \texttt{hasPrimaryActivity}\},$$
$$\{?property \rightarrow \texttt{bus} : \texttt{hasSecondaryActivity}\}\}.$$

The answer $\texttt{ans}(t_{cp}, DAB)$ of the query is the set of following triples.

```
bus:hasFormat rdfs:domain bus:Company .
```

Algorithm 1: TripleExtraction

Input: *endpoint*, *batchsize*

Output: *tripleset*

begin

1 $P \leftarrow retrieveDistinctProperty(endpoint)$

2 **foreach** *property* $p \in P$ **do**

3 $x \leftarrow retrieveNumberOfTriples(p, endpoint)$;

4 **for** $count \leftarrow 0$ **to** $x/batchsize$ **do**

5 $tripleset \leftarrow$
$tripleset + retrieveTriples(count, batchsize, endpoint)$;

6 **return** *tripleset*

```
bus:officialAddress rdfs:domain bus:Company .
bus:postalAddress rdfs:domain bus:Company .
bus:hasPrimaryActivity rdfs:domain bus:Company .
bus:hasSecondaryActivity rdfs:domain bus:Company .
```

Typically, data from a `Semantic Data Source` is retrieved by applying queries to a local RDF file or through a SPARQL endpoint. When the output of a given query is very big or if it is required to extract all triples from the `Semantic Data Source`, a simple download strategy might fail because some SPARQL endpoints restrict result sizes; DBpedia [38], for instance, does not allow more than 10,000 result triples at a time. To overcome this limitation, we design Algorithm 1. The algorithm takes a *sparqlEndpoint* URL and a *batchsize* (indicating how many triples to extract at a time) as parameters and returns the extracted triples. In line 1, the algorithm extracts all distinct properties in the KB. Then, for each property (lines 2-3), the algorithm determines the number of triples containing the property as predicate. Based on this number the algorithm determines how many queries need to be sent to receive all these triples and executes the queries (lines 4-5). This is done by sorting the triples by subject, using *batchsize* in the LIMIT clause, and using the iteration number multiplied by the batchsize as

OFFSET.

Extraction from Non Semantic Data Source.

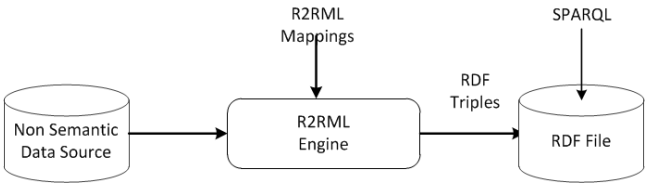


Figure 12: The process of extraction from `Non Semantic Data Source`.

Figure 12 shows the process of extraction from a `Non Semantic Data Source`. At first, we make a semantic version of the `Non Semantic Data Source` using the *R2RML Engine*. The *R2RML Engine* component is typically an R2RML processor that takes a relational database and an R2RML mapping document as input and outputs an RDF graph according to the mapping document [39]. Hence, the data of the `Non Semantic Data Source` are converted into RDF dataset which can simply be queried using SPARQL queries discussed in the Section 6.1. The following triples are the sample output of the *R2ML Engine* component with the input Listing 1 and Table 1.

```
sub:subsidy/10611390 sub:subsidyId 10611390 .
sub:subsidy/10611390 sub:paidTo sub:recipient/366894 .
sub:subsidy/10611390 sub:amount 617308 .
sub:subsidy/10600402 sub:paidTo sub:recipient/362146 .
sub:subsidy/10600402 sub:amount 6010 .
.....
```

6.2. Transformation

In the data transformation process, the extracted source data are transformed in a way such that it can easily be fed to the SDW. To ensure that data in the SDW are consistent with the semantics of its TBox, i.e., *ABox* follows *TBox*, we divide the transformation tasks into two Components: *Traditional Transformation*, and *Semantic Transformation*. The following sections detail the components.

Traditional Transformation. *Traditional Transformation* includes operations known from traditional ETL tools, such as cleansing the data and formatting the source data according to the target schema. This includes removing duplicate data, recalculating data, normalizing data, renaming attributes, checking integrity constraints, refining data, unique identifier generation, creating new attributes based on existing attributes, null value and default value handling, noisy data filtering, sorting data, and grouping/summarizing data [40]. In this component, the data related to each concept of the target TBox are stored in a separate table where the columns of the table represents the properties associated with the concept and the records of the table represent the instances of the concept. The transformed data are kept into a `Staging Area`. The `Staging Area` is an intermediate storage where the intermediate results of each process are stored. This can prevent the loss of transformed data in case of the failure of the loading process.

Semantic Transformation. *Semantic Transformation* includes operations to create RDF triples according to the semantics of the target TBox from the data output by the *Traditional Transformation* component.

Algorithm 2 describes the steps of converting a table (*table*) (from *Traditional Transformation*) into a set of RDF triples ($\mathbb{T}$). As additional inputs, the algorithm takes the name of an attribute in the table ($resourceKey$) that can be used to uniquely identify each row, a target TBox (*onto*), mapping files between the sources and the target (*mapping*), a *provenance graph* (*provGraph*) and $dataset_{name}$ is the name of the dataset to be created. It is an instance of `qb:Dataset`. The values of $resourceKey$ will be used to create resource identifiers. The *provGraph* is required to search for an existing IRI and to store the information of how the IRIs are formulated and its original source (literal/IRI and dataset) information. We describe the *Provenance Graph* in Section 6.3.

At first, a dictionary D is created based on the input *table* (line 1). D consists of (key, value) pairs for each row r in *table*; the key corresponds to r's $resourceKey$ value. The value in turn corresponds to a list of (attribute, value) pairs where each pair represents one of r's cells.

Algorithm 2: createTriples

Input: $table$, $resourceKey$, $onto$, $mapping$, $provGraph$, $dataset_{name}$

Output: $\mathbb{T}$

begin

```
1  D ← makeDictionary(table)
2  concepts ← getMatchingConcepts(table,mapping, concepts(onto))
3  foreach c ∈ concepts do
4      foreach r ∈ D do
5          sub ← createInstanceIRI(c, resourceKey(r), provGraph)
6          if !(propertyTable(table)) then
7              obj ← createConceptIRI(c, provGraph)
8              T.addtriple(sub, rdf:type, obj)
9              if (type(c) = qb4o:LevelProperty) then
10                 T.addtriple(sub, qb4o:memberOf, obj)
11             if (type(c) = qb:Observation) then
12                 T.addtriple(sub, rdf:type, qb:Observation)
13                 T.addtriple(sub, qb:dataSet, dataset_name)
14         foreach (attribute, value) ∈ r do
15             if (value! = NULL) then
16                 prop ← getMatchingProperty(attribute,
17                 mapping, property(c, onto))
18                 T.addTriple(sub, createPropertyIRI(c, prop, provGraph),
                   createObject(value))
19 return T
```

The next step is to determine which concepts in the target TBox the information contained in the table correspond to (line 2). Based on this information, the algorithm runs through each such concept c. Some tables in relational databases do not directly correspond to a concept, instead they represent many-

to-many relationships between instances of concepts (entity types). Hence, only for those tables that directly correspond to a concept, the algorithm creates a new resource for each entry in the dictionary D and a triple (with `rdf:type` as predicate) that defines the new resource as an instance of the concept (lines 6-8). To create an instance IRI, the *createInstanceIRI()* function (in line 5) first checks the *Provenance Graph* whether there is an existing IRI for the instance; if it finds an existing one, then returns it, else creates new one and updates the *Provenance Graph*. Different type of resources (i.e, concept, property, instance), takes different parameters to create new IRIs. Table 4 shows the parameters to create different type of resources.

Table 4: Required parameters for creating different types of IRIs

Type of IRI	Base IRIs	Concept Name	Property Name	Property Range Name	Value(s)	Resource Key(s)
Concept IRI	Y	Y				
Property IRI	Y	Y	Y			
Object Property value IRI	Y			Y	Y	
Instance IRI	Y	Y				Y

If the concept c is a `qb4o:LevelProperty`, then a triple saying that the new resource is a member of the level property c is added (lines 9-10). If the concept c is a factual concept, i.e., a `qb:Observation`, then the resource is added as a `qb:Observation` and the `qb:dataSet` of the resource is $dataset_{name}$ (lines 11-13). Furthermore, for each (*attribute*, *value*) pair for which the *attribute* is defined as a match of one of c's properties and *value* is not null, a triple encoding this information is created (lines 14-18). Depending on the type of property (data type or object), the object of the created triple either corresponds to a literal or a resource. For example, the values of `sdw:hasrecipient` are resources and the values of `sdw:amounteuro` are literals.

The computational complexity of Algorithm 2 depends on its constituents: the number of concepts in the target TBox matched with the input table, and the size of the given table. For each row of the input table, the *makeDictionary* operation creates an entry for the dictionary D. Therefore, line 1 takes $O(N)$

complexity, where N is the number of rows in the table. The loop depends on the number of concepts matched with the table. In the worst case, the number of concepts in the target TBox matched with the table is equivalent to the number of concepts in the target TBox. Let say C is the number of concepts in the target TBox matched with the input table. The for loop in line 4 is executed for each entry in D, therefore, it takes $O(N)$ time. The for loop in lines 14-18 is executed for each attribute of the table and it takes $O(M)$ time, where M is the number of attributes in table. Thus, the total time for Algorithm 2 is $O(N)+O(CNM) = O(CNM)$ in worst case, where C is the number of concepts in the target TBox, N is the size of the dictionary, and m is the average size of the entries of the dictionary. The best case is $O(NM)$ where, $C = 1$.

6.3. Provenance Graph

The provenance graph is an RDF graph that stores the meta information of concepts, properties, instances and IRIs of resources used in a SDW. At this version, we only consider the provenance of IRIs, and the provenance graph contains the type of resource it indicates in the target, its prefix (base IRI), its source dataset, and the original IRI or the original literal corresponding to each IRI. The type of a resource is either a concept, a property or an instance. For example, an IRI can be generated for a concept *Company*, for a property *hasCompany*, or for an instance of Company DanskBank. The prefix is the base IRI of the target. The graph is queried using SPARQL query. Figure 13 shows how an IRI is represented in the IRI provenance graph. The blank node represents that the IRI is an instance of `sdw:Day`. The literal used to generate the IRI is shown as the value of `pro:originalLiteral` which is taken from the `subsidy` dataset. The URL of the `subsidy` dataset are shown as the value of `pro:sourcedataset`. The new generated IRI is shown as the value of the `pro:generatedIRI` property. This graph needs to be built by semantic-ETL processes.

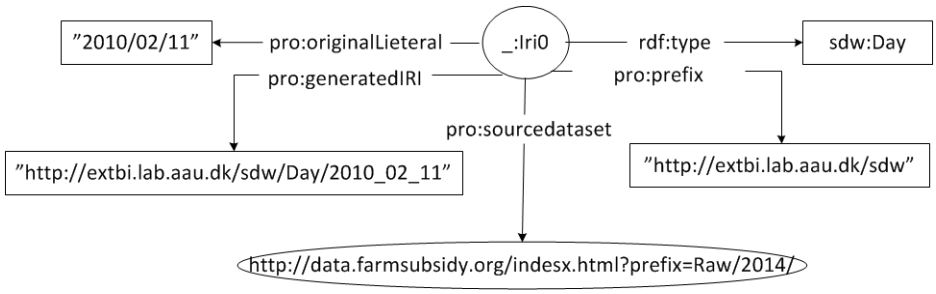


Figure 13: An IRI represented in the provenance graph.

Algorithm 3: LinkToExternalResources

Input: $intResource, externalDataSource, flag, k$

Output: $\mathbb{L}$

begin

1 $sb_{internal} \leftarrow semanticBag(intResource)$

2 **if** $flag = 0$ **then**
  $E \leftarrow search(intResource, externalDataSource, k)$

3 **else**
  $E \leftarrow search(intResource, externalDataSource, query, k)$

4 **foreach** $extResource \in E$ **do**

5 $extTriples \leftarrow retrieveTriples(extResource)$

6 $sb_{external} \leftarrow semanticBag(extTriples)$

7 **if** $match(sb_{internal}, sb_{external}) > \delta$ **then**

8 $alignedPairs \leftarrow alignedPairs \cup (intResource, extResource)$

9 $alignedPairs \leftarrow userInteraction(alignedPairs)$

10 **foreach** $pair \in alignedPairs$ **do**

11 $\mathbb{L}.addTriple(intResource, \texttt{owl:sameAs}, extResource)$

12 $ER \leftarrow getEquivalentResource(extResource)$

13 **foreach** $equiResource \in ER$ **do**

14 $\mathbb{L}.addTriple(intResource, \texttt{owl:sameAs}, equiResource)$

15 **return** $\mathbb{L}$

6.4. External linking

Algorithm 3 formalizes the steps required to link an internal resource to external resources. It takes *intResource*, *externalDataSource* *flag*, and k as input parameters. *intResource* is a given internal resource that we want to find external links for, and *externalDataSource* can be either a keyword search API or the SPARQL endpoint of a KB. In our DOLAP workshop paper [15], we use the Sindice API [41] as a default API, However, the service of the API is currently stopped by its provider. Therefore, in this version, we replace the default API with the DBpedia Lookup API [2]). *flag* indicates whether the target external source is a search API ($flag = 0$) or an external KB ($flag = 1$), and k is the number of top k IRIs that we want to retrieve.

At first, the algorithm creates a semantic bag ($sb_{internal}$) for the internal resource (line 1). Such a semantic bag consists of triples describing the internal resource. Then, if $externalDataSource$ is a search API, then a web service request embedding the $intResource$ is sent through the API for the top k matching external resources (line 2). If $externalDataSource$ is the SPARQL endpoint of a KB, then it submits a query to the endpoint for retrieving top k matching external resources (line 3). An example query is shown in Listing 2. The query retrieves top-20 resources from a KB which are the subjects of triples whose objects are literals containing the string "obama" most. For each external resource ($extResource$), the algorithm retrieves triples describing it (line 5), creates a semantic bag ($sb_{external}$) from them (line 6), and compares the bag to the one created for the internal resource (line 7). If the Jaccard Simarity (see Equation 1) between the two semantic bags exceeds a certain threshold δ, the internal and external resource are considered a match (lines 7-8).

$$J(sb_{internal}, sb_{external}) = |\frac{sb_{internal} \cap sb_{external}}{sb_{internal} \cup sb_{external}}| \quad (1)$$

After having identified all candidate pairs, the user can optionally interact with

[2] https://github.com/dbpedia/lookup

the system and filter pairs. Then, for each pair of internal and external resources, a triple with the `owl:sameAs` property[3] is created to materialize the link in the dataset (line 11). Finally, the algorithm also retrieves the set of resources from the KB that are already linked to the external resource via by `owl:sameAs` property and materializes the links to the internal resource as well (lines 12-14).

The computational complexity of Algorithm 3 depends on its constituents: the generation of semantic bags, the similarity computation, and the number of aligned pairs. A semantic bag of a resource is created from the RDF graph of the resource [42, 43]. The set of triples describing the resource defines the RDF graph of that resource. Therefore, a semantic bag creation takes $O(N)$ time complexity where N is the size of the graph. Hence, line 1 takes $O(N)$ time. The given internal source is compared to every external source. Therefore, the loop in line 4 is executed $O(K)$ times where K is the number of external sources, and since each *semanticBag* operation takes $O(N)$ time, the total time for all calls to *semanticBag* is $O(NK)$. The *match* operation matches the semantic bags of two resources and takes $O(N^2)$ time. Therefore, the total time of the loop lines 4-8 is $O(KN + KN^2) = O(KN^2)$. The maximum number of aligned pairs can be the equivalent of the number of external sources K. Hence, the total time for the loop in lines 10- 14 is $O(KR)$, where R is the average number of equivalent resources of each aligned external sources. Thus, the total time for Algorithm 3 is $O(N + KN^2 + KR) = O(K(N^2 + R))$, where K, N, R are the number of external resources, N is the average size of the RDF graphs, and R is the average number of resources same as with external resources.

Listing 2:**An example SPARQL query for keyword based searching**

```
1 SELECT ?sub (count(?sub) as ?count)
2 WHERE {
3    ?sub rdf:type ?class.
```

[3]An `owl:sameAs` property indicates that two IRI references refer to the same real world object. Hence, subject and object IRIs of a triple with `owl:sameAs` are considered linked.

```
    ?sub ?pro ?label.
    Filter regex(?label, "obama", "i")
}
group by ?sub
order by desc(?count)
limit 20
```

6.5. Load

This component loads the RDF triples created by *Semantic Transformation* into a triple store or saves to an RDF dump file. SETL currently uses Jena TDB [44] as a triple store and allows loading data batch-wise. It provides two kinds of load, namely, trickle load and bulk load. The trickle load mode transforms and feeds data as it arrives from the previous transformation step without staging on disk using SPARQL INSERT queries. As it provides concurrent processing and loading, therefore, it takes long time to store whole dataset because it includes process and loading time. In bulk load mode, the triples are written into a file on disk after transformation and then loaded batch-wise into the triple store.

7. Implementation

We implement the SETL framework, discussed in Sections 4, 5, 6, using Python [45]. The reasons for choosing Python are its comprehensive standard libraries and its support to programmer productivity [46]. The following sections discuss how we implement the main components of the framework.

7.1. Definition Layer

SDW TBox Definition . We define three Python Classes: *Concept*, *Property*, *BlankNode* to give the structure of concepts, properties and blank nodes of a TBox, respectively. These classes play a meta modeling role and the user can

instantiate the classes to define their TBox constructs, such as, concept, property, and blank node. The user can define the MD constructs, such as, cube, dimensions, level, factual concept, and hierarchies by instantiating the *Concept* class. Listing 3 shows a segment of Python script to create a TBox of our running example. It shows how to define different MD constructs, such as, dimension (lines 1-4), level (lines 5-7), hierarchy (lines 8-11), hierarchy step (lines 12-17), level attributes (lines 18-27), measure (lines 28-32), cube structure (lines 33-40), cube (lines 41-43), factual concept(43-44). To internally connect each concept with an explicit set of properties, such as data type property, level attribute, object property, functional property, inverse-functional property, SETL offers a method *conceptPropertyBinding()* which takes the list of all concepts, properties and blank nodes as parameters (line 51). The method *createTriples()* creates the triples according to the TBox (line 53).

Listing 3:**A segment of Python script to create the TBox of our running example**

```
1  # Defining Time Dimension
2  Time=Concept(name='PayDate', _base_iri='sdw:')
3  Time.setrdfType(['owl:class','qb:DimensionProperty'])
4  Time.setqb4oHasHierarchy('sub:TimeHierarchy')
5  Time.setrdfsLabel('''PayDate"@en')
6  # Defining levels for Time Dimension
7  Day=Concept(name='Day', _base_iri='sdw:')
8  Day.setrdfType(['owl:class','qb4o:LevelProperty'])
9  Day.setrdfsLabel('''Day"@en')
10 Month=Concept(name='Month', _base_iri='sdw:')
11 Day.setrdfType(['owl:class','qb4o:LevelProperty'])
12 Day.setrdfsLabel('''Month"@en')
13 # Defining a time hierarchy
14 TimeHier=Concept(name='TimeHier', _base_iri='sdw:')
15 TimeHier.setrdfType(['owl:class','qb4o:Hierarchy'])
16 TimeHier.setqb4oHasLevel(['sdw:day','sdw:Month'}}
```

```
TimeHier.setqb4oInHierarchy(['b_t1'}}
# Defining a hierarcy step
b_t1=BlankNode(name='_:t1')
b_t1.setrdfType('qb4o:HierarchyStep')
b_t1.setqb4oparentLevel('sub:Month')
b_t1.setqb4ochildLevel('sub:Day')
b_t1.setqb4opcCardinality('OneToMany')
b_t1.setqb4oinHierarchy('sdw:TimeHier')
# Defining level attributes
hasMonth=Property(name='hasMonth',_base_iri='sdw:')
hasMonth.setrdfType(['owl:ObjectProperty',
'qb4o:LevelAttribute'])
hasMonth.setrdfsRange('sdw:Month')
hasMonth.setrdfsDomain('sdw:Day')
paydate=Property(name='paydate',_base_iri='paydate')
paydate.setrdfType(['owl:objectProperty',
'qb4o:levelAttribute'])
paydate.setrdfsDomain('sdw:Subsidy')
paydate.setrdfsRange('sdw:Day')
amount=Property(name='amounteuro',_base_iri='sdw')
amount.setrdfType(['owl:datatypeProperty',
'qb:MeasureProperty'])
amount.setrdfsDomain('sdw:Subsidy')
amount.setrdfsRange('xsd:double')
# Defining a data structure definition
cubestruct=Concept(name='cubestruct',_base_iri='sdw:')
cubestruct.setrdfType(['owl:Class',
'qb:DataStructureDefinition'])
cubestruct.setqbComponent(['b_d','b_m'])
b_d=BlankNode(name='_d')
```

```
b_d.setqbdimension('sdw:Time')
b_m=BlankNode(name='_m')
b_m.setqbmeasure('sdw:amount')
# Defining a cube
cube=Concept(name='dataset',_base_iri='sdw:')
cube.setrdfType(['owl:Class','qb:DataSet'])
cube.setqbstructure('sdw:cubestruct')
# Defining factual concept
Subsidy=Concept(name='Subsidy',_base_iri='sdw:')
Subsidy.setrdfType(['owl:Class','qb:Observation'})
# Defining list for concepts, properties, and blank node
conceptList=list()
propertylist=list()
blankList=list()
# Extending the lists
conceptList.extend([Time,Day,TimeHier])
propertyList.extend([hasMonth,amount,paydate])
blankList.extend([b_t1,b_d,b_m])
# Establishing relationships among the elements
conceptPropertyBinding(conceptList,propertyList,blankList)
# Generating triples for the tbox
createTriple(conceptlist,propertyList,BlankList)
```

We also define a Python class named *ParseOntology*(*object*) to parse a given TBox. The user can instantiate it by passing the path of the given TBox. It provides several methods (*getConcepts(), getDataProperties(), getProperties()*, and etc.) to process the TBox.

Define Mapping. To define the mappings between the elements (concepts and properties) of a source TBox and the target TBox, we use a Python *dictionary* which is composed of (key, value) pairs. The key of a pair is the source element and the value of the pair is a list which contains the corresponding

target element and relationship type (lines 1-8 in Listing 4). The relationship type is presented using *sup*, *sub*, and *equi*. Here, *sup* means that the source concept is the super class of the target concept, *sub* means source concept is the sub class of the target concept, and *equi* means *equivalence* relationship. We also define the mappings between all source TBoxes and the target TBoxusing a Python *dictionary* whose keys are the source TBoxes names and the values are the individual source and target mapping dictionary (lines 9-10). Currently, SETL supports basic mapping, however, in the future we will allow it to support complex mappings, for example, mapping a property of a source TBox to a concept of the target TBox.

Listing 4: **Python script for defining mapping**

```
1  bus_sdw=dict()
2  bus_sdw={'bus:Owner':['sdw:Recipient', sup],
3  'bus:Activity':['sdw:mainActivity', 'sup'],
4  'bus:Activity':['sdw:SecondaryActivity', 'sup'],
5  'bus:hasCompany':['sdw:hasCompany','equi'],
6  'bus:hasPrimaryActivity':['sdw:hasPrimaryActivity','equi']
7  ,'bus:BusinessFormat':['sdw:BusinessType','equi'],
8  'bus:hasFormat': ['sdw:inCompany','equi']}
9  mapping=dict()
10 mapping['bus']=bus_sdw
```

For processing a `Non Semantic Data Source`, users can define a TBox representing the source using the classes of *SDW TBox Definition*. Then, the mapping between the target and the extracted TBox can be defined using Python *dictionary*. For converting the `Non Semantic Data Source` into RDF triples based on R2RML mapping, we implement the simple algorithms of R2RML processor provided in [39].

7.2. ETL Layer

SETL enables the extraction of data from a KB through SPARQL endpoints, RDF files, relational databases, CSV files, and shapefiles. The framework is designed in a way that allows it to be easily extended to support other formats.

Extraction from a SPARQL endpoint. Using the SPARQLWrapper [47] library, we extract data/triples from a KB by applying queries through a SPARQL endpoint.

Extraction from an RDF file. We use the RDFLib library [48] to execute a query on a local RDF file. If the RDF file is too large to fit into main memory, we split the file into several smaller files, execute the query on the smaller files, and then combine the partial results. However, this method can only be used for simply queries, such as retrieving all triples with a particular predicate. More complex queries involving joins need to be handled differently, e.g., by loading the data into a local SPARQL endpoint first and using the method described in the previous paragraph.

Extraction from a relational database or a CSV file. SETL is based on Petl [40], which is a traditional ETL Python package including a rich set of methods to perform extraction, cleansing, and transformation based on relational databases and CSV files. Making use of these methods, SETL provides the functionality as well.

Extraction from a shapefile. To read data from a shapefile, we use the pyshape [49] library. Internally, SETL can store the data of a shapefile either in a file or in a database table. SETL provides methods to automatically create a table in a database based on the structure of a shapefile and to insert the data from the shapefile into the created table. It also provides a method to store the attribute information of a shapefile in CSV format.

Figure 1 shows that there is no direct connection between *Field* and *OrganicField*, neither between *Field* and *FieldBlock*. In the target TBox (Figure 3), however, we have defined connections, e.g., `agri:Field` and `agri:OrganicField` are connected through the `owl:sameAs` property and `agri:FieldBlock` and `agri:Field` are connected via the `agri:contains` property. These connections

are computed based on a spatial join that SETL computes during the extraction process for pairs of shapefiles. We use the ArcPy [50] library to perform the spatial join analysis.

Transformation. To perform the operations related to *Traditional Transformation*, described in the 2nd paragraph of Section 6, we again use the Petl [40] library. For creating RDF triples according to the target TBox based on the data output from the *Traditional Transformation* component, we develop a Python method, named *createTriples(table, resourceKey, onto, mapping, provGraph, filepath)* that implements Algorithm 2, where the parameters maintain the same meanings as Algorithm 2. Additionally, it requires a parameter $filepath$, the location of the file, to store the generated RDF triples.

External Linking. We develop a method named *link(intResource, extSource, flag, k)* that implements Algorithm 3. We use Python built-in *requests* library to retrieve the top-k related resources of *inResource* through DBpedia Loopup API in JSON format. Sometimes the queries submitted to a SPARQL endpoint result in timeout error. For this reason, we download the RDF dump files of the selected KB and store them to a triple store locally for accessing it through local SPARQL endpoint.

Load. Bulk load stores the RDF triples from a file (created by $createTriple$ () method) to the triple store. To support bulk load, SETL offers two methods: *bulkTDBLoader (tdblocation, filePath, batchsize)* and *bulkSPARQLInsert(tdblocation, filePath, batchsize, database)*. The *BulkTDBLoader* is used to load data from a file on disk into a triple store using the jena TDBLoader command. The parameter *tdblocation* denotes the location of a TDB database, the *filePath* is the path to the triple file, and *batchsize* is the number of triples to be fed into the store at a time. The *BulkSPARQLInsert* is similar to *BulkTDBLoader* but defines SPARQL INSERT statements to load triples instead of using the jena TDBLoader command. It additionally requires the name of the database as parameter. Unlike bulk load, which loads the triples from a file, trickle load converts the cleansed and transformed data into triples and subsequently loads them into a triple store holding the triples in main memory. The

trickleload(*table*, *resourceKey*, *onto*, *mapping*, *provGraph*, *tdblocation*, *batchsize*) method has same parameters like *createTriple*() method. In addition, it takes *tdblocation* and *batchsize* as input parameters to directly load the triples into the triple store.

8. Evaluation

To evaluate SETL, as discussed in Section 3, we first create a SDW called SETLKB applying the proposed ETL process on sources covering Danish agricultural and business information. Then, we create an MD SDW called SETLSDW using SETL by integrating SETLKB and the `Subsidy` dataset. We create SETLKB in order to be able to compare it with an existing dataset integrating DAD and DBD [19] that has been created in a manual process by using a broad range of different tools. In contrast, SETL allows to perform this process mostly automatically with minimal user interaction. This allows us to compare the different ETL processes as well as the RDF datasets themselves. Furthermore, to show the strength of SETL over the non-semantic traditional data integration tools, we also re-create SETLSDW using Pentaho Data Integration (PDI, also called Kettle) [51] and compare the productivity and the performance of the processes with SETL.

Table 5 shows the description of SETLKB and SETLSDW. As DAD and DBD do not have any numerical measures, we do not model SETLKB in an MD fashion. On the other hand, SETLSDW is defined with MD constructs. It has 4,392,390 facts, 2 dimensions, 9 hierarchies, and 15 levels.

In the following, we refer to the dataset published by [19] as ExtBIKB. Our evaluation concentrates on three aspects: productivity, i.e., to what extent SETL eases the work of a user when performing an ETL task to produce a SDW, quality of the produced datasets, and performance, i.e., the time required to run the processes.

We run the experiment on a HP ProLiant DL385 server with an AMD Opteron(tm) processor 6376 with 32 cores (only 1 core is used in the experi-

ments); it has 256 GB DDR3 RAM and is running Ubuntu 14.04.1 LTS (Trusty Tahr). The data is stored on an 1-TB SCSI disk running in an HP Smart Array.

Table 5: Description of SETLKB and SETLDW

Dataset	# of RDF Triples	# of Concepts	# of Datatype Properties	# of Object Properties	# of External Linkings	# of Dimensions	# of Levels	# of Level Attributes	# of Hierarchies	# of Observations
SETLKB	34,177,652	16	76	22	14,153	-	-	-	-	-
SETLSDW	54,995,678	54	63	16	3,538	2	15	58	9	4,392,390

8.1. Productivity

One of the main advantages of working with SETL is that all the phases of an ETL process can easily be maintained and implemented using Python. SETL offers high-level classes and functionality for development of semantic ETL processes. Using SETL the users can write their own code to create an ETL process in Python. Therefore, it allows extensibility. The modularized structure of SETL allows users to call different modules simply by passing arguments. Hence, it hides the complexity of the processes from the users. For example, to populate the SDW with transformed data, the user just write one line of code to call the method *createTriples(table, resourceKey, onto, mapping, provGraph, filepath)* by passing the required arguments, where the parameters maintain the same meanings as Algorithm 2. Concretely, SETL uses 224 lines of code to implement Algorithm 2. Hence, it provides a higher level abstraction to the user by hiding details. Table 6 summarizes the tools, languages, and the Lines Of Code (LOC) used to produce the SETLKB dataset using SETL and the ExtBIKB dataset using the process described in [19]. To process a shapefile, SETL provides methods to automatically create and insert data into PostgreSQL. It uses only 2 lines of Python code. To perform spatial intersection operation between two shapefiles, SETL uses the ArcPy [50] library, which is faster than ArcGIS [52] which is used in [19]. On the other hand, the ETL process used in [19] converts the shapefile into PostgreSQL table using the PostgreSQL tool *shp2pgsql*. To make it compatible with vtSQL, they have performed some find-replace actions. For performing, spatial operation, they

use ArcGIS [52]. For filtering inconsistent values, unknown characters and for correcting candidate keys, we use different methods from the Petl library [40], which are efficient and easy to access. To cleanse our all dataset, SETL only takes 173 lines of code. On the other hand, the [19] uses different MySQL views and hand-crafted methods to cleanse the data, and it takes 360 LOC. To generate triples for all data sources, SETL takes only 109 LOC whereas [19] uses 1,511 LOC. SETL only takes two line of codes to call the external linking procedure. On the hand, [19] does not support any linking process. The last row of Table 7 shows the total number of LOC. SETL takes in total 570 LOC to make the ETL process where 286 LOC is used to build the TBox. On the other hand, for ExtBI, we sum only the LOC of *Cleansing* and *Triple Generation* because we can not count the LOC for other processes, hence, the sum is $1879 + 5NA$. In summary, SETL is significantly more productive, using only a fraction of the LOC of $1879 + 5NA$ and using only Python instead of 7 different languages and tools.

PDI contains a rich set of data integration functionality to create an ETL solution for non-semantic data sources. However, it does not support any functionality to support semantic integration. We use PDI in combination with other tools and manual tasks to re-create SETLSDW. Table 7 shows the comparison between the ETL processes of SETL and PDI to create SETLSDW. SETL provides built-in classes to annotate MD constructs with a TBox. On the other hand, PDI does not support to define a TBox. To create SETLSDW using PDI, we use the TBox created by SETL. SETL 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 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. The class also includes the functionality to internally connects the concepts and properties of the TBox. PDI is a non-semantic data integration tools, thus, it does not support to process semantic data. In this case, we manually extract data from SPARQL endpoints and materialize them in a relational

database to further processing. PDI provides drag & drop functionality to process database and CSV files. On the other hand, SETL provides methods to extract semantic data either from a SPARQL endpoint or an RDF dump file batch-wise. SETL allows users to create an IRI by simply passing argument to the *createIRI ()* method. Moreover, SETL supports to update the provenance graph to store the provenance information of IRIs. On the other hand, PDI does not include any functionality to create IRIs for resources. We define a 23 line of Java class to enable the creation of IRIs for resources. To generate triples from the source data based on the MD semantics of the target TBox, SETL provides *createTriple()* method; users can just call it by passing required arguments. On the other hand, 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. Finally, we could not run the ETL process of PDI automatically (i.e., in a single pass) to create the SETLSDW. We make it with significant number of user interactions. In total SETL takes 401 LOC to run the ETL, where PDI takes $471LOC + 4NA + 21Activity$. Thus, SETL creates an MD SDW with minimum number of LOC, user interactions comparing to PDI where users have to build their own Java class, plugin, and manual tasks to enable semantic integration.

Table 6: Comparison between the ETL processs of SETL and [19]

Tools	**SETL**			**ExtBI**		
Task	**Used tools**	**Used language**	**LOC**	**Used tools**	**Used language**	**LOC**
TBox Definition	Built-in SETL	Python	286	Virtuoso, Protege	Not Applicable (NA)	NA
CSV	Built-in Petl	Python	2	MySQL,vtSQL	SQL	NA
Reading Shapefile	Built-in SETL	Python	3	ArcGIS,PostgreSQL, shp2pgsql	NA	NA
Spatial join	Built-in ArcPy	Python	2	ArcGIS	NA	NA
Cleansing	Built-in Petl	Python	173	MySQL, Google refine	SQL	360
Triple Generation	Built-in SETL	Python	2	Virtuoso	iSQL	1511
External Linking	Built-in SETL	Python	2	NA	NA	NA
Loading	Built-in SETL	Python	1	Virtuoso	iSQL	4
Total LOC for the complete ETL process			**570**	**1879 + 5 NA**[4]		

[4]Note that this is not the total number of LOC for the complete ETL process because

Table 7: Comparison between the ETL processes of SETL and PDI for SETLSDW

Tools	SETL			PDI (Kettle)		
Task	**Used Tools**	**Used Lnaguages**	**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 Activity: 1
Cleansing	Built-in Petl	Python	36	Drag & Drop	NA	Number of used Activity: 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 Activity: 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 Activity**		

8.2. Quality

In this section, we examine the extent to which the produced triples are consistent, complete, interpretable, fresh, and semantically linked [53]. We compare the result created with SETL (SETLKB) to ExtBIKB.

Consistency. According to the definition, an object property relates instances of two classes, i.e, if x is a value of an object property, there must exist at least one triple whose subject is x. We found out that for a significant number of values for the object properties `bus:postalAddress` and `bus:officialAddress`, ExtBIKB does not contain such information. There are 83,302 `bus:officialAddresses` and 34 `bus:postalAddresses` that are not used as a subject of any triple in ExtBIKB. In SETLKB, on the other hand, all `bus:officialAddresses` and `bus:postalAddresses` have correctly been related to the `bus:Address` concept. Hence, data consistency is completely maintained in SETLKB.

ExtBI uses different tools for different processes. Thus, it could not run the ETL process in a single pass.

Information loss. In ExtBIKB, fields that do not produce any crop are not represented. This affects 2,650 instances of the concept `agri:Field`. Similarly, instances of `agri:OrganicField` that have the value "999999-99" for the `agri:FieldBlock` property are not represented in ExtBIKB. This affects 1,213 instances. For SETLKB on the contrary, we do not want to miss any `agri:Field` or `agri:OrganicFields` instance even though they do not have crop information. Hence, in the cleansing step, we replace the value "999999-99" with *NULL*. If an instance of a concept contains *NULL* or an unknown value for a property, the *createTriples ()* method (described in Section 6) simply does not produce a triple for that property. This is how SETLKB ensures data completeness.

In the original raw datasets, we can find out since when a legal unit belongs to an owner. This is not possible in ExtBIKB if an owner owns more than one company. We fix this issue by extending the target TBox. In addition, we also add the information related to the type of the owner, i.e., whether the owner is a person or another company, both are missing in ExtBIKB.

Furthermore, the main and secondary activity of a company or production unit are treated as the same activity in the ExtBIKB. In SETLKB, the concepts `bus:Company` and `bus:ProductionUnit` are connected to `bus:Activity` through two object properties: `bus:hasMainAcitivity` and `bus:hasSecondary Activity`. Thus, SETLKB provides more information than ExtBIKB.

Redundant triples. If a property is inversely connected to another property, then storing the same types of triples for both properties, simply by interchanging the subject and predicate of the triples, makes a KB redundant. The properties `bus:belongsTo` and `bus:hasProductionUnit` are inversely connected to each other and relate the concepts `bus:Company` and `bus:ProductionUnit`. In ExtBIKB, triples are stored for both properties separately, which produces extra 659,639 triples. On the other hand, we create triples only for `bus:belongsTo` property. Hence, SETLKB reduces redundancy.

External linking. We enhance SETLKB by linking it to external knowledge bases at instance level. Section 6.4 describes the linking process. On the

other hand, ExtBIKB is not connected with other external knowledge bases at instance level. We run the linking process only for instances of `agri:Crop`, `agri:FieldBlockApplication`, `bus:BusinessFormat`, and `bus:Municipality` because for instances of other concepts, DBpedia [38] does not contain relevant information. Note that the following results are produced by the automatic matching process without any pre-filtering by the user (which is mentioned as an optional step in Section 6.4). Table 8 shows the different types of concepts for which we run the linking process, their total number of internal instances, the number of internal instances linked to the external resources, the number of external resources connected to the internal instances, and the accuracy rate of the linking. In total, 196 instances of SETLKB are linked to 9,618 external resources. To evaluate the linking process, we randomly chose 5% of the triples that linked internal and external resources and manually evaluated their correctness. The accuracy rate in Table 8 shows the accuracy rate in percentage. We only consider the values of `rdfs:comment`, `rdfs:label` and `rdf:description` in the semantic bags of compared resources in order to make the process faster. One of the reasons having the low accuracy rate for some cases is the language. The description of each instance is translated from Danish to English using Google Translate API, which sometimes fails to give accurate translation.

Listing 5:**The SPARQL query that connects SETLKB with DBpedia to retrieve the description of top-3 municipalities where most of the companies belongs to.**

```
PREFIX bus:<http://extbi.lab.aau.dk/ontology/business/>
PREFIX dbo:<http://dbpedia.org/ontology/>
PREFIX owl:<http://www.w3.org/2002/07/owl#>

select ?municipality ?abstract
where {
        {select ?municipality count(*)
        where {
        ?company a bus:Company.
        ?company bus:officialAddress ?address.
```

```
        ?address bus:positionedAt ?addFeature.
        ?addFeature bus:inMunicipality ?municipality.}
Group by ?municipality
order by desc(count(*))
limit 3
}
?municipality owl:sameAs ?db.
SERVICE <http://dbpedia.org/sparql?default-graph-uri=
  http://dbpedia.org> { ?db dbo:abstract ?abstract .
  FILTER (lang(?abstract) = 'en')}
}
```

Because of being linked with external resources, users can analyze SETLKB based on external knowledge bases as well. Listing 5 shows an SPARQL query that retrieves the descriptions of top-3 Danish municipalities from DBpedia with the most companies in them. The query returns eight entries. The first 3 entries describe Copenhagen, next 2 entries describe Aarhus, and the last 3 entries describe Aalborg. This query cannot be answered by ExtBIKB.

Table 8: Linking local resources to external resources

Concept	**# of instances**	**Linked instances**	**External resources**	**Accuracy**
agri:Crop	244	95	5,309	83%
bus:BusinessFormat	30	16	976	66%
agri:FieldBlockApplication	17	6	403	43%
bus:Municipality	98	79	2,930	73%
Total	**389**	**196**	**9,618**	**66.25%**

Based on these findings, we can conclude that SETLKB is more consistent, complete, and resourceful than ExtBIKB. Table 9 summarizes some notable triple-wise differences between SETLKB and ExtBIKB.

Table 9: Comparison between SETLKB and ExtBIKB

Concept(c)/ Property(p)	# of Triples SETLKB	# of Triples ExtBIKB	Comments
Whole KB	34,177,652	32,457,657	SETLKB is the updated version.
External Linking	9,618	0	No instance is linked with external sources in ExtBIKB.
Company(c)	603,667	603,668	One duplicate triple in ExtBIKB.
Address(c)	499,850	497,938	In ExtBIKB, Address concept were not mapped with the postal address of Danish Business dataset.
Field(c)	52,060	50,842	ExtBIKB used poor cleansing techniques.
OrganicField(c)	613,903	611,093	ExtBIKB used poor cleansing techniques.
belongsTo(p)	659,638	659,640	Two duplicate triples in ExtBIKB.
hasProductionUnit(p)	0	659,640	belongsTo and hasProductionUnit are inversely connected.
hasOwnerType(p)	353,954	0	ExtBIKB did not include owner type information.

8.3. Performance

In this section, we discuss the time required for different sub-processes to create SETLKB and SETLSDW using SETL, the performance of different sub-process, and the comparison of performance with the previous solution [19] and PDI.

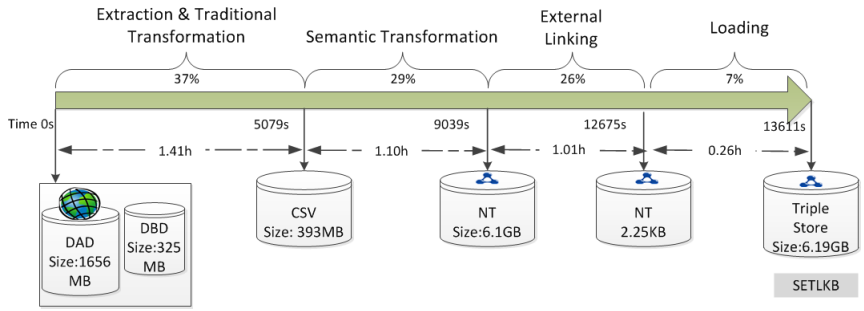


Figure 14: Time for the ETL process to create SETLKB. s and h stand for second and hour respectively.

Runtime of the different phases. Figure 14 and 15 illustrate the time and the percentage of the total time that each phase of the ETL processes of SETLKB and SETLSDW respectively accumulated. The figures show both the time for each phase in separate and the overall runtime of the complete process. The times represent averages over 5 test runs. As SETL automatically connects all the phases, the end time of one phase becomes the starting time of the next.

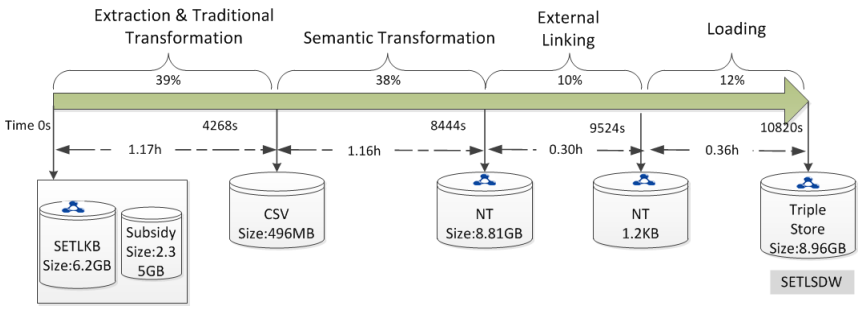


Figure 15: Time for the ETL process to create SETLSDW

Figure 14 and 15 also show the materialized output with the data size of each phase.

Figure 14 shows that the *Extraction* and *Traditional Transformation* phase take 37% of the total time. This phase takes the longest time comparing to other phases because of the following reasons. First, the DAD is given with three Shapefiles: *Field*, *Organic Field*, and *Field Block*; therefore, we need to do some preprocessing tasks to extract the attribute information from the files and to keep them into a PostgreSQL database. Second, we perform two spatial join operations: between *Field* and *Field Block*, and between *Field* and *Organic Field* datasets. To perform the spatial join operations, we use ArcPy library which takes 30% time of this phase which is out of our control. Third, we extract a TBox from the dataset and based on the source-target mappings, we keep the data into 9 smaller tables. We perform extensive cleansing operations for correcting candidate keys, fixing inconsistent and incomplete data, and filtering noisy data which are also time intensive. Fourth, we also fix and filter the noisy, inconsistent and incomplete data from Company and Participant datasets and split the two datasets into 19 smaller tables according to the source-target mappings which are also time consuming.

The *Semantic Transformation* process is time-intensive because all the data needs to be processed and matched to the TBox of the SETLKB. In Section 6.2, we discuss the computational complexity of *Semantic Transformation* which depends on the three nested loops. The first loop iterates for the number of

concepts in the target TBox matched with the input table. The second loop repeats for each row of the input table. Further, each iteration creates an IRI for each row of the table and updates the `Provenance Graph`. The third loop iterates for each value of a row, and *Semantic Transformation* creates a separate triple for each value in the SDW according to the semantics encoded in the target TBox. On the other hand, the building block of a traditional DW is tuple which contain multiple values. Many triples of SDW are kept into a tuple of a traditional DW table. This is why our SDW takes longer time comparing to the traditional DW.

The total time required for the *Linking* process is 3,657 seconds. All the information of the resources are given in Danish, so we first translate the name of the resource into English, then use DBpedia Lookup API to retrieve the top 20 relevant resources. Eventually, we use our instance matcher described in Section 6.4 to link the internal and external resources. The computational complexity depends on the number of external resources to be matched, the average size of the RDF graphs of internal and external resources, and the number of resources same as external sources. As we consider the semantic bags of two resources in the matching operation, we need to retrieve the RDF graph of each external resource from DBpedia through its SPARQL endpoint. Moreover, the matching operation is time consuming because it takes $O(N^2)$ complexity, where N is the average size of the RDF graphs of internal and external resources. Table 10 shows the time consumed for linking the instances of each concept. In summary, SETLKB is linked to 9,618 external resources. To load all triples at a time, the *BulkTDBLoader* takes 936 seconds which is almost 7% of the total time. For loading triples into a triplestore, we depend on the native loading commands of the underlying triplestore. The elapsed time depends on the internal strategy of the triplestore which is also out of control. We have discussed the performance of the different loading methods in the earlier workshop paper [15].

In Figure 15, it is shown that the ETL process takes SETLKB and the `Subsidy` dataset as input. the *Extraction* and *Traditional Transformation* phase

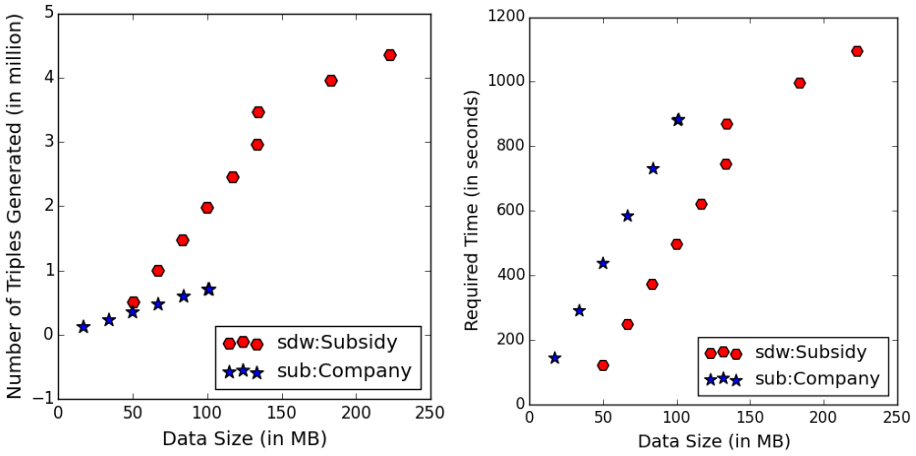
includes time for 1) Filtering the noisy data from the `Subsidy` dataset (2.3 GB in size), 2) converting the `Subsidy` dataset into RDF according to the R2RML mapping, 3) extracting data from the `Semantic Data Sources` through its SPARQL endpoints, and 4) stores them into the `Staging Area` based on the source-target mappings. The *Semantic Transformation* makes the RDF triples based on the MD semantics encoded in the target. Here, we only link the instances of `sdw:Municipality` with the external resources. In total, the ETL process for creating SETLSDW takes 10,820 seconds.

Table 10: Time required for the linking process

Concept	Time(in seconds)
`agri:Crop`	1,925
`bus:BusinessFormat`	504
`agri:FieldBlockApplication`	63
`bus:Municipality`	1,165
Total	**3,657**

Performance analysis of Semantic Transformation. Figure 16 shows the performance of our *Semantic Transformation* method. To evaluate the method, we consider the factual concept `sdw:Subsidy` from SETLSDW and the central concept `sub:Company` from SETLKB because `sub:Company` also acts as a super class of the `sdw:Company` level of SETLSDW. Figure 16a shows the number of triples generated with the increasing input data size, and Figure 16b shows the processing time with the increasing input data size. With the increase of input data size, the number of generated triples (shown in Figure 16a) and the processing time (shown in Figure 16b) increase almost linearly for both concepts. In general, the processing time and the number of generated triples depends on the nature of the input data, e.g., whether it corresponds to a fact entry, a dimension entry or a level entry, how big the value of each attribute is. Figure 16 summarizes that the number of triples generated per data size and the data processing rate for `sub:Company` are higher than that of `sdw:Subsidy`.

This is because each entry of `sdw:Subsidy` only contains a measure property and the pointers to two dimensions, namely, `sdw:Beneficiary` and `sdw:Time`. Therefore, `sdw:Subsidy` is straightforward to process while `sub:Company` requires more effort as it involves 17 properties with relatively long strings, such as address, name, email, etc. Table 11 reports the performance of *Semantic Transformation* for `sdw:Subsidy` and `sub:Company` using different comparison metrics. Triple generate rate of *Semantic Transformation* for `sdw:Subsidy` is 4.9 times higher than that of `sub:Company`. `sdw:Subsidy` has a higher data process rate (i.e., the amount of data that can be processed per time) than `sub:Company`. To process 1 MB input data, `sdw:Subsidy` takes 4.8 seconds while `sub:Company` takes 8.7 seconds. We also compare the number of triples per (1 MB) data size for both input and output data size. 1 MB of the output file generated for `sdw:subsidy` contains 6,134 triples where `sub:Company` contains 5,393 triples. The output file size for `sdw:Subsidy` is 31 times larger than the input data size. On the other hand, the output file for `sub:Company` is 12 times larger than its input file.



(a) Number of triples vs data size. **(b)** Processing time of increasing data.

Figure 16: Performance of *Semantic Transformation* based on the number of generated triples and processing time with increasing data size.

Table 11: Comparison to process the factual concept and a level of the running example by *Semantic Transformation*

Comparison Metrics	**sdw:Subsidy**	**sub:Company**
Triple generate rate (triples/second)	39816.47	8089.80
Data process rate (data size/second)	0.22MB	0.11MB
Data process time (time to process unit data size (in MB))	4.80s	8.70s
Average time for generating a triple	25.20μs	120 μs
Number of triples per input unit data size (in MB)	190986.40	69961.85
Number of triples per output data size (in MB)	6134	5393
Output data size and input data size ratio	31	12

Comparison between SETL and ExtBI [19]. To make it comparable with the results of [19], we also run the computer on our local machine which is 2.10 GHz Intel Core i7-2600 processor with 8 GB RAM operated by Windows 7 and similar to the machine used by ExtBI. Table 12 shows the time in seconds takes different sub-processes of the ETL processes. In this case, we do not run the *External Linking* process. SETL takes more time in data cleansing step comparing to ExtBI. This is because we handle the null value and other unknown values which are simply discarded in ExtBI. On the other hand, our *createTriple()* method takes less time to make the RDF dump although we create 2M more triples than ExtBI shown in Table 9. Loading RDF triples more time as we load more triples than ExtBI. In summary, SETL takes 5,725.05 seconds where ExtBI takes 6,141.56 seconds, i.e., SETL takes 6.7% less time than ExtBI.

Table 12: Comparison between SETL and ExtBI [19]

Tools	Data Cleansing (excluding Spatial join)	Load Ontology	Load Mappings	Dump RDF	Load RDF	Total
ExtBI	603.35	1.01	12.35	4,684.82	840.04	6,141.56
SETL	726	1.05	0	4,208	970	5,725.05

Comparison between SETL and PDI in processing `sdw:Subsidy`. To compare the performance of SETL with a Non-semantic data integration tool, we choose PDI. We populate the SDW for `sdw:Subsidy` concept using PDI. Figure 17 shows the ETL flow for `sdw:Subsidy` built on PDI. The *TBox Extraction*, *R2RML Mapping*, *Extraction* using SPARQL steps are ignored in this flow as those sub-processes cannot be implemented using PDI, i.e., the ETL run with PDI starts by taking the Subsidy dataset transformed into a semantic source. To run the process PDI takes 1903 seconds. On other hand, SETL takes 2221 seconds to complete the process which includes all sub-processes starting from *TBox Extraction* to *Load*. If we run the ETL using SETL by skipping he sub-processes skipped by PDI, then SETL takes only 1644 seconds. SETL takes 577 seconds for the sub-processes *TBox Extraction*, *R2RML Mapping*, and *Ex-*

traction from semantic source. Thus, SETL is 259 seconds (13.5%) faster than PDI.

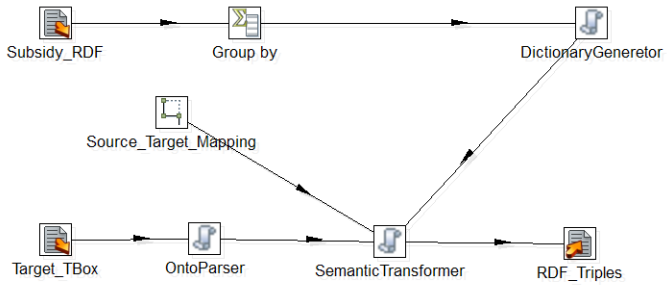


Figure 17: ETL flow of PDI to populate `sdw:Subsidy`

9. Related Work

This section discusses the state of the art in the area of semantic ETL to build a semantic DW and to integrate and publish data as Linked Open Data (LOD) from multiple sources.

Recently, the use of SW technology in data warehouses has become popular. Some approaches have used ontologies as a data integration method for mapping between sources and target whereas others have considered a data warehouse as a repository of ontologies and its instances.

In [1, 12], an OWL ontology is used to link the schemas of semi-structured and structured data to a target DW schema. At first, the schemas of the data sources and the DW are described by a common graph-based model named datastore graph. Then, an application ontology is constructed to describe the semantics of the data sources and the DW, and mapping between them can be done through that ontology. In this way, the heterogeneity issues among the schemas of sources and target have been resolved. In this work, it is assumed that source and target schemas are known beforehand. However, it does not consider the MD view over the DW. In [54], the authors have shown that an ontological approach can automate and minimize the maintenance cost of the

evolution of a DW schema. Here, they define every design step of DW using ontology.

A framework for designing semantic DW has been proposed in [55]. Here, the usage of SW languages to integrate distributed DWs and to automate the ETL process of a DW has been discussed. The DW has been considered as a repository of ontologies and semantically annotated data sources. [56] proposes a methodology describing important steps required to create a semantic DW that enables to integrate data from semantic databases, which is composed of an ontology and its instances. A multidimensional (MD) framework has been proposed in [57] to analyze semantic data that are provided by the SW and that are annotated by application and domain ontologies. It allows to build and populate a DW from both the analyst requirements and the knowledge encoded in the ontologies. In [30], authors have proposed a solution for supporting data integration tasks by constructing ontologies from XML and relational sources and integrating the derived ontologies by means of existing ontology alignment and merging techniques. However, ontology alignment technique itself is a difficult and error-prone process [58].

In the past couple of years, publishing data in a machine-readable format has become more popular and (Linked Open Data) LOD has emerged as a way to share such data across Web sources. [59] presents a process to publish governmental data as LOD and [19] discusses how to spatially integrate and publish a Danish Agricultural dataset and a Danish Business dataset as LOD. The approach uses views to cleanse the data and Virtuoso for creating and storing RDF data. To integrate data from heterogeneous sources and publish those data as LOD, a semantic ETL framework is presented in [8] at conceptual level. The approach uses the semantic technology to facilitate the integration process and discusses the use of different tools to accomplish the different steps of the ETL process. To enable warehouse-style analysis over RDF data, some approaches [60, 61] propose the use of a native RDF data warehouse. To analyze an RDF graph, [60] introduced a lens named analytical schema, which is a graph of classes and properties. Each node of the schema represents a set of facts,

which can be analyzed by exploring the reachable nodes. To enable OLAP-style analysis over Linked Data (LD), W3C recommend RDF Data Cube Vocabulary (QB vocabulary) to describe the RDF data in MD fashion [28]. However, it has limitations to fully define the traditional MD constructs and it is mostly used to present the statistical data as LD. In [14], authors propose QB4OLAP model by extending the QB model to annotate an existing graph with MD constructs. To enable OLAP over LD datasets which contains numerical values, even if they are not annotated either QB or QB4OLAP, in [62], authors proposed a SPARQL-based ETL framework.

Because of the decentralized structure and dynamic nature of LOD, sometimes it is required to integrate external sources in a federated way. Unlike DW, data federation avoids the need of materializing integrated data into a centralized warehouse. It emphasizes on logical rather than physical integration of data. In [63], authors have proposed a theoretically well-founded approach to the logical federation of OLAP and XML data sources. A set of query processing strategies for executing OLAP-like SPARQL queries over a federation of SPARQL endpoint has been proposed in [64]. However, federation techniques have an additional number of unique challenges on several levels. Participating sources in a federation are autonomous, i.e., they might be unavailable at some point or change their data. So, for query processing, federations introduce delays because of the distribution and there is no guarantee that the data is still available or in the same state as when the federation was created - both schema and data might have changed so that the integration rules might no longer be valid. So, a standard approach is to avoid federations and have a local copy of the data which is the focus of this study. This guarantees that the data will be available and an answer to a user query can be computed efficiently. Moreover, scalability is indeed a problem for analytical queries in federations [64].

As data become outdated due to updates in the sources, using different tools for extraction, validation, and integration becomes very tedious and time consuming. Hence, a framework is necessary that makes it possible to accomplish every step in a single platform. Although the state-of-the-art approaches pro-

vide different conceptual frameworks, there is no implemented programmable framework that facilitates the developers by providing required tools, classes, and methods to build ETL process with the goal of either creating a semantic multidimensional DW or publishing the semantically integrated data as LOD. Hence, this paper discusses a semantic ETL framework, SETL, which provides various modules, classes, and methods to extract semantic-aware data, geospatial data, and other traditional data, to integrate source data semantically using a target TBox defined with/without MD constructs, and to store the data as semantic data. SETL facilitates developers to build a semantic DW using a single platform instead of using different tools and a manual process.

10. Conclusion

Building a DW integrating internal and external data published in different formats requires semantic integration. Here, we have proposed and developed a programmable framework, named Semantic ETL (SETL) that facilitates users to build a (MD) SDW. SETL uses TBox as an underlying schema to integrate heterogeneous data sources. To annotate the target TBox with MD constructs, it uses the constructs of QB4OLAP model. It allows to process both `semantic` and `Non Semantic Data Sources`. In order to process a `Non Semantic Data Source`, it builds a semantic layer on top of the `Non Semantic Data Source`. Eventually, it stores the data as RDF triples based on the MD semantics encoded in the target TBox. It also allows to link the internal resources with external resources. In order to evaluate our framework, we produced a SDW by integrating a `semantic` and a `Non Semantic Data Sources`. In Section 8, we show that SETL has good performance for all the steps and is faster than the competing tools/solutions.

The model behind a triplestore is an RDF graph. Another issue is how to implement physically the store. We can choose graph, relational or Hadoop, but in this paper, we focus on increasing the abstraction level and thus the user productivity. Therefore, physical or platform-dependent optimizations remain

for the future. Our future research also includes proposing a well-defined set of basic semantic ETL operators that can be combined to perform any semantic ETL operations. Besides, developing techniques to explore the Linked Open Data cloud for the related dimensions and level and linking them with the internal ones are also in our consideration.

Acknowledgments

This research has been funded in part by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC).

References

[1] D. Skoutas, A. Simitsis, Designing ETL Processes using Semantic Web Technologies, in: DOLAP, 2006, pp. 67–74.

[2] C. S. Jensen, T. B. Pedersen, C. Thomsen, Multidimensional databases and data warehousing, Synthesis Lectures on Data Management 2 (1) (2010) 1–111.

[3] A. Abelló, O. Romero, T. B. Pedersen, R. Berlanga, V. Nebot, M. J. Aramburu, A. Simitsis, Using Semantic Web Technologies for Exploratory OLAP: A Survey, TKDE 99 (2014) 571–588.

[4] W3C. Resource Description Framework, `http://www.w3.org/RDF/`.

[5] R. Berlanga, Ó. Romero Moral, A. Simitsis, V. Nebot, T. Pedersen, A. Abelló Gamazo, M. J. Aramburu, Semantic web technologies for business intelligence.

[6] M. S. Kotoulas, F. van Harmelen, J. Weaver, Kr and reasoning on the semantic web: Web-scale reasoning, in: Handbook of Semantic Web Technologies, Springer, 2011, pp. 441–466.

[7] K. Christodoulou, N. W. Paton, A. A. Fernandes, Structure inference for linked data sources using clustering, in: Transactions on Large-Scale Data- and Knowledge-Centered Systems XIX, Springer, 2015, pp. 1–25.

[8] S. K. Bansal, Towards a Semantic Extract-Transform -Load (ETL) Framework for Big Data Integration, in: Big Data, 2014, pp. 522–529.

[9] M. E. Zorrilla, J.-N. Mazón, Ó. Ferrández, I. Garrigós, F. Daniel, J. Trujillo, Business Intelligence Applications and the Web: Models, Systems and Technologies, Business Science Reference, 2012.

[10] W3C. Resource Description Framework Schema, `http://www.w3.org/TR/rdf-schema/`.

[11] OWL Web Ontology Language, `www.w3.org/TR/owl-ref/`.

[12] D. Skoutas, A. Simitsis, Ontology-based Conceptual Design of ETL Processes for both Structured and Semi-structured Data, IJSWIS 3 (4) (2007) 1–24.

[13] F. Baader, The description logic handbook: Theory, implementation and applications, Cambridge university press, 2003.

[14] L. Etcheverry, A. Vaisman, E. Zimányi, Modeling and Querying Data Warehouses on the Semantic Web Using QB4OLAP, in: DaWak, 2014, pp. 45–56.

[15] R. P. Deb Nath, K. Hose, T. B. Pedersen, Towards a programmable semantic extract-transform-load framework for semantic data warehouses, in: Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP, ACM, 2015, pp. 15–24.

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

[17] Ministry of Food, Agriculture and Fisheries of Denmark, http://en.fvm.dk/.

[18] Danish Central Company Registry (CVR) Data, http://cvr.dk/.

[19] A. B. Andersen, N. Gür, K. Hose, K. A. Jakobsen, T. B. Pedersen, Publishing Danish Agricultural Government Data as Semantic Web Data, in: JIST, 2015.

[20] A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J. N. Mazón López, F. Naumann, T. B. Pedersen, S. Rizzi, J. C. Trujillo Mondéjar, P. Vassiliadis, et al., Fusion cubes: towards self-service business intelligence.

[21] ESRI, Shapefile Technical Description, INC, 1998.

[22] M. Golfarelli, S. Rizzi, Data warehouse design: Modern principles and methodologies, McGraw-Hill, Inc., 2009.

[23] A. Nasiri, E. Zimányi, R. Wrembel, Requirements engineering for data warehouses., in: EDA, 2015, pp. 49–64.

[24] A. Abelló, J. Samos, F. Saltor, Yam2: a multidimensional conceptual model extending uml, Information Systems 31 (6) (2006) 541–567.

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

[26] R. Kimball, M. Ross, The data warehouse toolkit: the complete guide to dimensional modeling, John Wiley & Sons, 2011.

[27] L. Etcheverry, S. S. Gomez, A. Vaisman, Modeling and querying data cubes on the semantic web, arXiv preprint arXiv:1512.06080.

[28] R. Cyganiak, D. Reynolds, J. Tennison, The rdf data cube vocabulary, World Wide Web Consortium.

[29] C. Bizer, A. Schultz, The berlin sparql benchmark (2009).

[30] R. Touma, O. Romero, P. Jovanovic, Supporting data integration tasks with semi-automatic ontology construction, in: Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP, ACM, 2015, pp. 89–98.

[31] A. D. H. Thi, B. T. Nguyen, A semantic approach towards cwm-based etl processes, Proceedings of I-SEMANTICS 8 (2008) 58–66.

[32] M. Ehrig, Ontology alignment: Bridging the semantic gap, volume 4 of semantic web and beyond computing for human experience (2007).

[33] J. Li, J. Tang, Y. Li, Q. Luo, Rimom: A dynamic multistrategy ontology alignment framework, Knowledge and Data Engineering, IEEE Transactions on 21 (8) (2009) 1218–1232.

[34] G. Petasis, V. Karkaletsis, G. Paliouras, A. Krithara, E. Zavitsanos, Ontology population and enrichment: State of the art, in: Knowledge-driven multimedia information extraction and ontology evolution, Springer-Verlag, 2011, pp. 134–166.

[35] R2RML: RDB to RDF Mapping Language `https://www.w3.org/TR/2012/REC-r2rml-20120927/`.

[36] J. Sequeda, F. Priyatna, B. Villazón-Terrazas, Relational database to rdf mapping patterns., in: WOP, Citeseer, 2012.

[37] E. V. Kostylev, J. L. Reutter, M. Ugarte, Construct queries in sparql, in: LIPIcs-Leibniz International Proceedings in Informatics, Vol. 31, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[38] DBpedia, `http://dbpedia.org/`.

[39] S. Zhou, Z. Xu, Y. Ni, H. Zhang, R2rml processor for materializing rdf view of relational data: algorithms and experiments, in: Web Information System and Application Conference (WISA), 2013 10th, IEEE, 2013, pp. 450–455.

[40] Petl - Extract, Transform and Load (Tables of Data), https://petl.readthedocs.org/en/latest/.

[41] G. Tummarello, R. Delbru, E. Oren, Sindice. com: Weaving the open linked data, in: ISWC, 2007.

[42] M. H. Seddiqui, S. Das, I. Ahmed, R. P. D. Nath, 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.

[43] M. Seddiqui, R. P. D. Nath, M. Aono, et al., An efficient metric of automatic weight generation for properties in instance matching technique, arXiv preprint arXiv:1502.03556.

[44] Apache Jena TDB, https://jena.apache.org/documentation/tdb/.

[45] Python, python.org as of 2015-03-25.

[46] C. Thomsen, 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, ACM, 2009, pp. 49–56.

[47] SPARQLWrapper, http://rdflib.github.io/sparqlwrapper/.

[48] Rdflib, https://github.com/RDFLib/rdflib.

[49] Pyshp, https://code.google.com/p/pyshp/.

[50] ArcPy-ArcGIS Python Library, http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#//000v00000001000000.

[51] M. Casters, R. Bouman, J. Van Dongen, Pentaho Kettle solutions: building open source ETL solutions with Pentaho Data Integration, John Wiley & Sons, 2010.

[52] Esri, Arcgis. http://www.esri.com/software/arcgis.

[53] V. Theodorou, A. Abelló, W. Lehner, Quality Measures for ETL Processes, in: DaWaK, 2014, pp. 9–22.

[54] M. Thenmozhi, K. Vivekanandan, An Ontological Approach to Handle Multidimensional Schema Evolution for Data Warehouse, IJDMS 6 (4) (2014) 33–52.

[55] V. Nebot, R. Berlanga, J. M. Pérez, M. J. Aramburu, T. B. Pedersen, Multidimensional integrated ontologies: a framework for designing semantic data warehouses, JDS XIII (2009) 1–36.

[56] L. Bellatreche, S. Khouri, N. Berkani, Semantic Data Warehouse Design: From ETL to Deployment à la Carte, in: DASFAA, 2013, pp. 64–83.

[57] V. Nebot, R. Berlanga, Building Data Warehouses with Semantic Web Data, DSS 52 (4) (2012) 853–868.

[58] P. Tyl, Ontology matching for web services composition, in: e-Technologies and Networks for Development, Springer, 2011, pp. 94–103.

[59] B. Villazón-Terrazas, L. M. Vilches-Blázquez, O. Corcho, A. Gómez-Pérez, A Methodological Guidelines for Publishing Government Linked Data, in: Linking government data, 2011, pp. 27–49.

[60] D. Colazzo, F. Goasdoué, I. Manolescu, A. Roatiş, RDF Analytics: Lenses over Semantic Graphs, in: WWW, 2014, pp. 467–478.

[61] K. A. Jakobsen, A. B. Andersen, K. Hose, T. B. Pedersen, Optimizing RDF Data Cubes for Efficient Processing of Analytical Queries, in: COLD, 2015.

[62] T. Komamizu, T. Amagasa, H. Kitagawa, Spool: a sparql-based etl framework for olap over linked data, in: Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services, ACM, 2015, p. 49.

[63] D. Pedersen, K. Riis, T. B. Pedersen, Xml-extended olap querying, in: Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on, IEEE, 2002, pp. 195–206.

[64] D. Ibragimov, K. Hose, T. B. Pedersen, E. Zimányi, Processing aggregate queries in a federation of sparql endpoints, in: European Semantic Web Conference, Springer, 2015, pp. 269–285.