IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Crowd Sourced Semantic Enrichment (CroSSE) for knowledge driven querying of digital resources

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1726444 since 2020-02-03T21:11:40Z

Published version:

DOI:10.1007/s10844-019-00559-8

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

29 March 2024

Noname manuscript No.
(will be inserted by the editor)

Crowd Sourced Semantic Enrichment (CroSSE) for Knowledge Driven Querying of Digital Resources

Giacomo Cavallo · Francesco Di Mauro · Paolo Pasteris · Maria Luisa Sapino · K. Selcuk Candan

Received: date / Accepted: date

Abstract Today, most information sources provide factual, objective knowledge, but they fail to capture personalized contextual knowledge which could be used to enrich the available factual data and contribute to their interpretation, *in the context of the knowledge of the user who queries the system.* This would require a knowledge framework which can accommodate both objective data and semantic enrichments that capture user provided knowledge associated to the factual data in the database. Unfortunately, most conventional DBMSs lack the flexibilities necessary (a) to prevent the data and metadata, evolve quickly with changing application requirements and (b) to capture user-provided and/or crowdsourced data and knowledge for more effective decision support. In this paper, we present CrowdSourced Semantic Enrichment (CroSSE) knowledge framework which allows traditional databases and semantic enrichment modules to coexist. CroSSE provides a novel Semantically Enriched SQL (SESQL) language to enrich SQL queries with information from a knowledge base containing semantic annotations. We describe CroSSE and SESQL with examples taken from our SmartGround EU project.

1 Introduction

When making decisions impacting public utility and encouraging and/or enforcing (possibly unpopular) behavioral rules, public administrators need to rely on data and knowledge supporting their choices, which can be used to better inform those citizens who will be affected by such decisions. While the advantages of bringing scientific data into a uniform integrated platform are clear, in this paper we note that this only solves part of the problem. We see that, in order to make the databank truly useful for a diverse group of researchers, analysts, and decision makers, the data needs to be complemented by one or more knowledge-bases that describe

Giacomo Cavallo, Francesco Di Mauro, Paolo Pasteris, Maria Luisa Sapino
Computer Science Department; U. Torino;
E-mail: {cavallo,dimauro,pasteris,mlsapino }@di.unito.it

K. Selçuk Candan
CIDSE; Arizona State University; E-mail: candan@asu.edu

the contexts in which the data is queried and explored. Moreover, given that it is not realistic to assume that tailor-made knowledge-bases will exist for all relevant contexts, it is critical that such knowledge (ontologies as well as assumptions and hypotheses) can be extended individually and collaboratively by the users.

In this paper, we present *Crowd Sourced Semantic Enrichment (CroSSE)*, a social knowledge platform and integrated services supporting semantic enrichment and content personalization within the context of scientific investigations. We note that CroSSE is domain independent (i.e., generalizable to many application domains), in that it can be used to annotate and semantically enrich any application database. More specifically, as we later discuss in Section 4, users of the semantic annotation tool can associate their semantic annotations and enrichments on any value on any attribute they find in the original database. While CroSSE platform is domain independent, in this paper, we describe our work within the context of the *SmartGround - SMART data collection and inteGRation platform to enhance availability and accessibility of data and infOrmation in the EU territory on SecoNDary Raw Materials* - European project, which aims to develop a databank platform providing access to a broad spectrum of data relevant to decision making in the context of waste collection and management.

1.1 Use Case: SmartGround

Both urban and mining waste contains materials that can still be useful. The challenge is to define rational waste management practices which would enable the re-use of that part of waste which could be recovered from industrial, mining, and municipal landfills[1]. The SmartGround platform integrates existing information from national and international databanks (national agencies, public bodies data bases, European statistics) and provides the data to all types of researchers and decision makers (city level, state level, European level) to perform hypothetical reasoning, possibly within different contexts, representing, for example, the rules and constraints enforced in different countries. In particular, decision makers may need to estimate the implications of actions they are considering, or the impacts of new laws (such as enforcing some prohibition, or setting new thresholds in the definition of allowed activities) which they are planning to enact (*"What would happen if no more than a certain amount of waste can be shipped to some specific landfill from a given region?"; "Would the available stocking site still be sufficient?"; "What if some combination of elements in a landfill was considered dangerous, and its presence would trigger a fine to the manager of the landfill?"; "How many landfills would be charged high fines?"*). Moreover, additional (and many times ad-hoc) assumptions that may vary from user to user or from location to location, may be provided as the context: *"Assuming that the presence of ⟨some combination of elements⟩ in a landfill might pollute the air in a ⟨certain number of kilometers⟩ in the neighbourhood, what would be the estimated polluted area, given the available data about the waste deposits?"*.

The SmartGround platform consists of two main modules: (a) A *main database* collects the data on the landfills in a relational database and provides the users with the tools to explore and update its content. (b) A *semantic platform* collects

[1] In Europe, there are from 150.000 to 500.000 very variable landfills. In 2008, 49% of the almost 3 billion tons of total waste generated in the EU-27 was disposed in dedicated landfills.

and manages the ontological information provided by the users, offering the tools to perform enriched queries on the main platform database (see Section 5). The knowledge base at the core of SmartGround consists of rules and ontologies that formally describe the relationships among key concepts at different levels of abstraction. Such knowledge potentially includes observational concepts related to context, sampling, classification, and measurement. Importantly, users are allowed to enter concepts into the knowledge base and to relate them to known concepts or concepts declared by other researchers. See Section 3 for more details.

1.2 Our Contributions: Semantic Tagging and Semantic Query Enrichment

In order to support the above, our Crowd Sourced Semantic Enrichment (CroSSE) system enables users to enrich the information stored in the databank with their own knowledge to personalise queries and reasoning tasks. For example the director of a specific laboratory might be interested in combining the information about the analysis on a landfill (information stored in the database), with other information relevant to her but not stored in the database (for example, the data and role in the laboratory of the person who has signed the analysis report), for *querying the database in the context of her personal knowledge*. To support such contextualised querying process, CroSSE provides a Semantic Tagging Module (see Section 4) in which users can insert their own knowledge (and possibly share knowledge already inserted and made available by other users). Such knowledge is represented within CroSSE in the form of RDF statements [4] and a query engine combines SPARQL queries on each user's knowledge base and SQL queries for the data stored in the relational databank. In [11], we discussed the need for semantic tagging and contextualised queries to enable crowdsourced participation to decision making processes, and we provided an overview of the semantics of the enriched queries we were targeting. In this paper, we introduce the syntax of the contextually enriched query language SESQL and the system architecture which enables SESQL queries.

2 Related Work

In many application domains, including sciences, there is a strong need to be able to collaborate through sharing of data, information, and knowledge. Data integration technologies and crowdsourcing platforms, such as MiNC [1] and LabBook [15], provide great opportunities in this direction.

Data Integration: In general, there are three types of information-integration systems. In source-centric systems, the sources are defined in terms of the global schema and are referred to as local-as-view, or LAV, systems (Information Manifold [16], Emerac [23]). The LAV approach, while flexible, assumes a consistent integrated view. An alternative approach is to define the global schema in terms of the sources. This is called global-as-view, or GAV (HERMES [2], SIMS [3], TSIMMIS [13]), and WEBBASE [9,10]). The third class is a hybrid referred to as a GLAV system [28]. Orchestra [25] and FICSR [7] are systems that focus on managing disagreements that arise (at both schema and instance levels) during data sharing. FICSR creates a data structure that captures all interpretations of a conflicting database and can provide different views, ranked with users individual assumptions and preferences, to different users. [29] provides a survey of different DB integration techniques.

Ontology Driven Query Formulation: In [12] authors discuss how reasoning on the ontology affects the query answering process in their Ontology-Based Data Access. ODBA can be implemented as a three level architecture consisting of the ontology, the data sources, and the mapping between them. The approach in [22] deals with ontology-driven query formulation, in which the intensional description of a relational database is mapped to a OWL-DL description, the language in which the domain experts express their specific knowledge. On this common OWL-DL formalization, the user may formulate ontological queries that are then translated into the corresponding relational SQL statements. Available ontologies can be used in web site management and integration scenarios; in particular, [18] describes a SEmantic portAL (SEAL) which presents a three-layer architecture encompassing: (a) *heterogeneous data sources* (DB, XML, HTML); (b) a *wrapper* that aggregates the sources in a common data model; (c) *integration modules* able to reconcile the data sources. The central aspect of this family of semantic portals (including SmartGround) is the help offered to a community of users, each one *contributing* to the global knowledge base while also *consuming* the common enriched knowledge. See [27] for a survey of relevant semantic technologies.

Processing of Semantic Queries: Ontologies describe intensional knowledge in which integrity constraints can be expressed in specific languages such as Description Logic and in particular the DL-Lite subset, that can guarantee a very efficient query answering process. In [6] a Tuple Generating Dependency (TGD) syntax for the rules describing ontological constraints is proposed, with some restrictions on variable occurrences in the body rules. Since it is difficult to express queries against complex ontologies, [17] describes a system that automatically infers the user's query from examples. A key problem in query rewriting is the expressive power of the rewriting scheme and the soundness and completeness of the supported query reformulation. A somewhat secondary, but important, problem is the optimization of the rewriting process: [26], for instance, supports semantic-aware inverted indexing, while in [14], the optimization step is achieved in terms of conjunctive queries against an ontology to obtain minimal output. The general problem of ontology based information retrieval is described in [21], not only in terms of relationship between ontology and relational database schema, but with a comparison between specific languages (RDF vs OWL1 vs OWL2), ontology-based tools and database to ontology mapping/transformation tools.

Ontology Management and CrowdSourcing: [20] and [19], focus on crowdsourcing ontology verification and engineering. They apply ontological verification to large biomedical ontologies, in which the class hierarchy not only is the core structure, but is the only semantic relationship created by ontology developers. Using a crowdsourcing method for ontology verification (in which workers answer computer-generated questions based on ontology axioms) the hierarchy verification is subdivided in micro-tasks and the results are measured. In [5] the problem of ontology based information reuse is oriented to the realization of knowledge-based digital ecosystems. The authors present techniques based on linguistic analysis that, starting from the vocabularies contained in each source ontology and relating them with the initial (or proto) ontology, can facilitate the process of ontology construction, automating the selection and reuse of existing data models. [24] presents the NeOn methodology for ontology engineering, which considers the ontological development as the construction of networks of ontologies, where resources may be managed by people in different organizations.

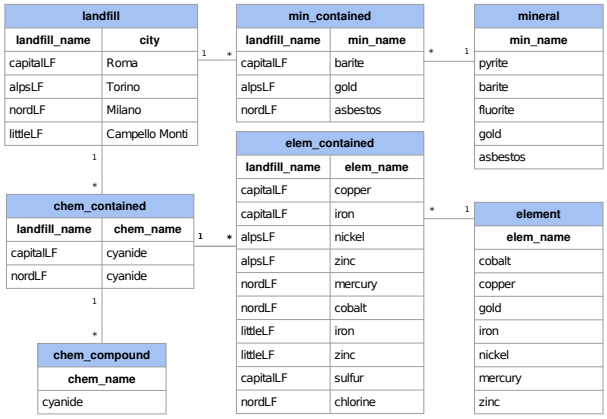


Fig. 1 Sample fragment of the SmartGround database: this segment of the database represents a sample data fragment capturing the knowledge of what is contained (in basic terms of *elements*, *minerals* and *chemical compounds*) in four mine landfills

3 Motivating Application: SmartGround European Project

Within the context of the *SmartGround*[2] European project, introduced in Section 1.1, we are developing a databank in which a broad spectrum of data relevant to decision making in the context of waste management are collected and shared. The databank integrates information from national and international databanks, public data bases, and European statistics. In the rest of this paper, we use a small subset of the SmartGround databank (see Figure 1) as a running example.

In this paper, we note that users of an integrated data store, like SmartGround, may often need to enrich the data stored in the databank with their own knowledge, to personalise queries and reasoning tasks. For example, a regulator body might be interested in combining the information about the content of a landfill (stored in the database) with other information relevant to them but not stored in the database (e.g., the hazardousness of the waste material in the landfill). To support such user participation, SmartGround leverages our Crowd Sourced Semantic Enrichment (CroSSE) system[3], which includes a *semantic tagging module* in which users can insert their own knowledge (and possibly share knowledge already inserted and made available by other users). This knowledge is represented in the form of an ontology, expressed in form of RDF statements [4].

Figure 2 provides a sample ontology capturing the classification of the possible types of waste that can be found in a mine landfill. Some of these concepts can be a direct reference to the database content (e.g., *mineral*, *element*, *chemical_compound*), while others can be part of the general common knowledge (e.g., geographical information) or can be user-defined concepts (e.g., *hazardous_waste*). The ontology concepts are connected by means of RDF properties which can be predefined (e.g. *type*, representing a parent-child relationship) or introduced (and potentially shared) by the users of the system (e.g., *ore_assemblage*). In this example scenario, the *ore_assemblage* property (ideally defined by a user who has

[2] http://www.smart-ground.eu/index.php

[3] The SmartGround platform, which implement many of the functionalities of the CroSSE, is available, for registered users, at *http://smartground.atosresearch.eu/home*.

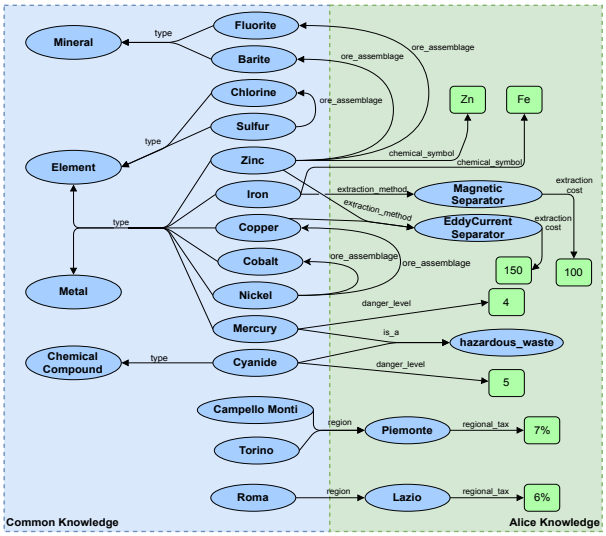


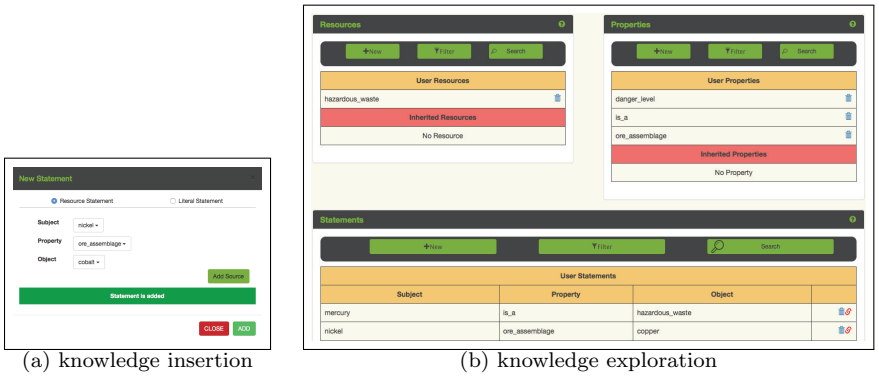
Fig. 2 Sample fragment of the SmartGround ontology

a geology background knowledge) associates to a given element/mineral the set of other elements/mineral found in the types of rock that may also contain the abovementioned element/mineral (e.g. *Nickel* can usually be found along *Cobalt* or *Copper*, see Figure 2). This information can be exploited to infer the content of a landfill even when it is not directly specified in the database. In Figure 2, we see that the knowledge base has two parts: a "common" part (shared by all users of the system) and a personalized part consisting of knowledge specific to a user named "Alice". We introduce these common and personalized semantic annotations next.

4 Semantic Tagging and Annotations

In CroSSE, we distinguish between (1) *data*, which are stored in the database and represent *factual information* shared by the different partner institutions and taken as certain knowledge by all the users, and (2) *personal, contextual knowledge*, which reflects the users' interpretation of the data, or the contextual meta-knowledge that the users might want to use in combination with the stored data.

The factual information, shared by all users, is stored in a (relational) database, DB. The knowledge base, KB, on the other hand, contains a set of user provided knowledge statements, which may or may not be shared. Intuitively, the knowledge in KB *enriches* (i.e, *contextualises* and *extends*) the information already available in the database and the conclusions that can be drawn from it. Each statement is annotated with information about its "source", the users who inserted it into the system and the users who have chosen to accept this statement as theirs. Given the set of all possible users, $\mathcal{U}$, the knowledge base can be logically seen as a mapping, $KB : \mathcal{U} \to \mathcal{P}(KB)$, which associates each user to the set of statements she believes in; i.e., $KB(u) = \{s = \langle subj, pred, obj \rangle \mid u \text{ believes in } s\}$. Given this, for the user, u, the knowledge she will rely on while querying the system will be $DB \cup KB(u)$. With a slight abuse of notation, we use KB for both the mapping which associates



(a) knowledge insertion (b) knowledge exploration

Fig. 3 SmartGround example: (a) inserting a knowledge statement to the user ontology; (b) exploring the user ontology

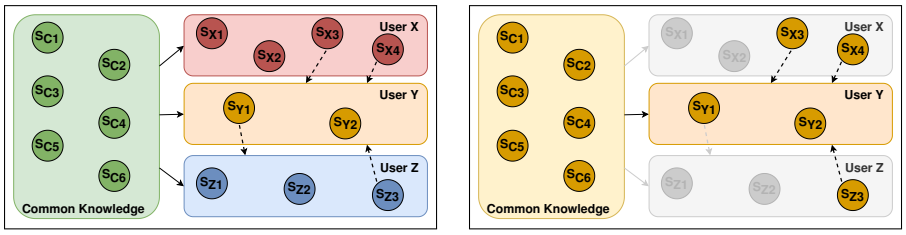


Fig. 4 (a) A knowledge base with information contributed by multiple users and (b) the contextualized knowledge base at query time

each user the statements she believes in and as the global set of available knowledge statements, with the intended meaning that $KB = \bigcup_{u \in \mathcal{U}} KB(u)$.

Scenario 1 (SmartGround Semantic Tags) Remember from Figure 1 that the SmartGround database includes data about the elements, minerals and/or chemical compounds that can be found in various landfills. The database, however, does not capture information about what elements (maybe if co-located with some others) might be considered as pollutant as this might depend on local (to the states or the regions) rules and regulations fixing thresholds for acceptable amounts of specific elements in space units. Yet, once semantically tagged, the users can query the SmartGround and obtain information about the existence of pollutant elements (extracted from the database) in regions of interest. Figure 3 shows insertion of a statement in the ontology and the knowledge exploration panel, respectively. ◊

The *semantic tagging module* of CroSSE enables users to extend the knowledge base three distinct ways: *(a) Integrated annotations:* Users can highlight a concept of interest and annotate it. Annotations can be of different nature: they can express properties about the concepts in the knowledge base and possibly used at query time or can be general notes the user is interested in storing for future use, for exploration purposes only. *(b) Independent annotations:* Users can directly access the semantic module and state their properties to be inserted in the knowledge base. *(c) Crowd-sourced annotation:* Users can explore the knowledge made available by their peers, and inherit relevant parts into their own knowledge base.

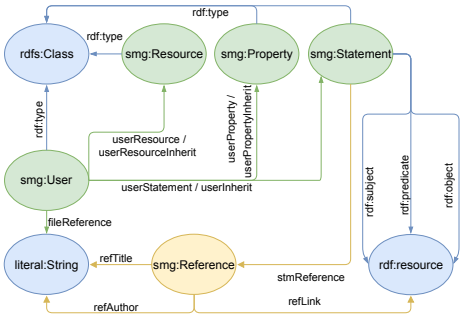


Fig. 5 CroSSE RDF triple store schema for semantic tagging

Scenario 2 (Knowledge Base) In Figure 4, each circle represents a statement, while the green box contains all the common statements related to concepts in the database. The other boxes contain the statements defined by three different users. In this example, the user Y has inherited the statements S_{X3} and S_{X4} from the user X (dotted arrows). When the user Y submits a Semantically Enriched SQL query, by default the system uses all her *personal* statements, as well as the statements that she *inherited* from other users and the *common* statements (see Figure 4(b)) to retrieve the contextual information. ◊

The knowledge statements are represented in form of RDF triplets: each statement is a triple $\langle subject, property, object \rangle$ meaning that the concepts associated to *subject* and *object* are related through the relationship *property*. For example $\langle mercury, is_a, hazardous_waste \rangle$ states that the element *mercury* belongs to the class *hazardous_waste*. Figure 5 illustrates the RDF schema CroSSE uses to represent the contextual knowledge. This schema allows the storage and querying of the contextual metadata (i.e., the RDF triples) while differentiating the ones defined by different users. Each statement, property, and resource is annotated (reified) with information about its "source", i.e. the user who inserted it into the system and the users who have chosen to inherit it. Each statement can also carry information about the reliability of the source or of the statement itself.

5 Semantically Enriched Query Processing

Query enrichment enables the users to exploit the semantic enrichments to obtain a more informative result set, which contains data derived by the common/shared data in the database, along with the one that is available to them in the knowledge base. In particular, CroSSE allows, through a novel SESQL query language (Figure 6), queries that can replace or extend the results from the database.

5.1 SESQL Overview

SESQL, briefly introduced in [8], offers clauses to help specify (a) the desired type of enrichment (either addition/removal of attributes in/from the query result table or use of ontological knowledge in the query filtering condition); (b) the attributes (from the relational schema) to be enriched; and (c) the ontological properties on which the enrichment has to be based. An SESQL query consists of two parts, the

```
terminal:
 ENRICH, SCHEMAEXTENSION, SCHEMAREPLACEMENT, REPLACECONSTANT,
 REPLACEVARIABLE, DEFINE, STRICT, KLEVEL, COMMON, PERSONAL, INHERITED,
 AS, STRING;

non terminal:
 body, expression, schemaext_exp, schemarep_exp, replacevar_exp,
 replaceconst_exp, klevel, scope, sparql_def, sparql_syntax,
 attribute, property, name, concept, s;

rules:
 s → ENRICH body [klevel] [sparql_def]
 klevel → KLEVEL(scope,scope,scope)
  | KLEVEL(scope,scope)
  | KLEVEL(scope)
 sparql_def → DEFINE query_name AS "[" sparql_query "]"
 body → [STRICT] expression body | [STRICT] expression
 expression → schemaext_exp | schemarep_exp | replaceconst_exp
  | replacevar_exp
 schemaext_exp →
  SCHEMAEXTENSION(attribute,property[,concept])[AS name]
  | SCHEMAEXTENSION(attribute,query_name,output_set)[AS name]
 schemarep_exp →
  SCHEMAREPLACEMENT(attribute,property[,concept])[AS name]
  | SCHEMAREPLACEMENT(attribute,query_name,output_set)[AS name]
 replaceconst_exp → REPLACECONSTANT(label,const,property)
  | REPLACECONSTANT(label,const,query_name)
 replacevar_exp  → REPLACEVARIABLE(label,attribute,property)
  | REPLACEVARIABLE(label,attribute,query_name)
 scope → COMMON | PERSONAL | INHERITED
 attribute → STRING                const → STRING
 property → STRING                 concept → STRING
 name → STRING                     label → STRING
 query_name → STRING               output_set → STRING | STRING output_set
 sparql_query → ... (defined according to sparql query syntax)
```

Fig. 6 BNF grammar for the SESQL language

first corresponding to a traditional SQL query and the second specifying the type of semantic enrichment the user is interested in:

```
SELECT ...    FROM ...    WHERE ...
ENRICH [STRICT]    ...    KLEVEL(...)
```

The `ENRICH` clause plays the role of the separator between the two query components, the standard SQL part and the enriching statements. In particular, SESQL enables sequences of enrichments of two distinct but complementary types:

- *enrichment by SELECT clause:* in this modality, the user leverages information retrieved from the ontology to *extend* or *replace* the values that the output tuples take for certain specified attributes;
- *enrichment by WHERE clause:* here, the user leverages the ontology to modify the condition specified in the `WHERE` clause; this enables the user to obtain the results she would have if the specified enrichment was implemented directly on the database before the execution of the `WHERE` clause.

In the case of enrichment by SELECT clause, the (optional) STRICT clause limits the query result set to the elements for which the ontological knowledge is explicitly expressed into the ontology. If omitted, the "closed world" assumption holds. In the case of enrichment by WHERE clause, the STRICT clause restricts the scope of the condition to only the information relevant in the context of the user's ontology (see Sections 6 and 7). The `KLEVEL` clause, on the other hand, enables the user to

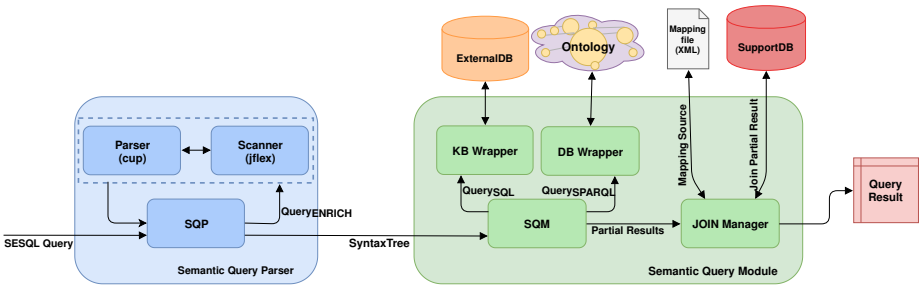


Fig. 7 CroSSE semantically enriched query processing

choose the desired scope for the query: in particular, the user, u, can query the set of RDF statements provided by herself ($KB_{\mathcal{U}}(x)$), a common ontology ($KB_{\mathcal{C}}$), the set of statements ($KB_{\mathcal{I}}(u)$) inherited from other users, or any combination. The user selects among these options by specifying one or more of the keywords "PERSONAL", "COMMON", and/or "INHERITED". If this clause is omitted, the default scope is the the union of all three scopes.

5.2 Semantically Enriched Query Processing Overview

Figure 7 presents the CroSSE module for semantically enriched query processing. The module has two sub components: the Semantic Query Parser (SQP) for the enriched query language and the Semantic Query Module (SQM) that queries the two relevant data sources and integrates the results. Given the syntax tree created by SQP from the given SESQL query, the SQM module creates a set of SPARQL queries to an RDF store to extract the relevant knowledge from the ontology, forming the necessary knowledge tables stored in a supplementary storage, *SupportDB*. An exception is raised if there are syntax errors in the SQL or SPARQL queries or if there are inconsistencies (e.g. the name of the knowledge table is not used in the enrichment expression). A *JOIN Manager* is used to combine the results the created knowledge tables in the SupportDB with data stored in the original database. A mapping file is used to resolve any naming differences between the original database and the knowledge tables in SupportDB. The *JOIN Manager* first formulates a *data-fetching* SQL query to be executed in the original database to fetch the relevant tuples from the original data source into temporary tables in the SupportDB. In order to generate the results of the SESQL query, the *Join Manager* next formulates a *data-merging* SQL query and executes it on the SupportDB to join the partial results from the database, stored in temporary tables, with the knowledge tables extracted from the ontology. In the rest of the paper, we formalize the Semanticaly Enriched SQL (SESQL) language contextually-relevant query formulation.

6 Enrichment by SELECT Clause

Let Q be a query with $A_1, \ldots, A_r$ attributes specified in the `SELECT` clause and let $RES(A_1, \ldots, A_r) = \{\langle a_1^1, \ldots, a_r^1\rangle, \ldots, \langle a_1^n, \ldots, a_r^n\rangle\}$ be the result of the SQL query, containing a set of n result tuples. Enrichment by `SELECT` clause operates on the results of the SQL query by replacing some of the returned values and/or

Table 1 Temporary tables and the result set for Example 1

elem_name	danger_level
mercury	4
cyanide	5

(a) KT

elem_name
mercury
cobalt
chlorine

(b) TT

elem_name	danger_level
mercury	4
cobalt	null
chlorine	null

(c) SESQL result

extending the schema. Let σ_i denote an *enrichment by* `SELECT` *clause* statement in the `ENRICH` clause. This statement is rewritten into a SPARQL query whose results are then represented in the form of a knowledge table KT_i with attributes $\mathcal{V}_{DB} \cup \mathcal{V}_{KB}$, where $\mathcal{V}_{DB}$ is a set of attributes from the database and $\mathcal{V}_{KB}$ is a set of attributes containing information extracted from the knowledge base:

- In *schema extension*, the result is extended with new knowledge attributes, combining factual data from the DB with ontological information from the KB, reported in the knowledge table; i.e., $RES' = RES \bowtie_{A_i \in \mathcal{V}_{DB}} KT$.
- In *schema replacement*, instead, the original factual values specified in A_i are replaced with the corresponding values from the KB, reported in the knowledge table; i.e., $RES' = \Pi_{(\{A_1,\ldots,A_r\}/\mathcal{V}_{DB}) \cup \mathcal{V}_{KB}} (RES \bowtie_{A_i \in \mathcal{V}_{DB}} KT)$.

Note that, as we will see in the examples presented in the rest of this section, the STRICT policy implementation requires the adoption of an *inner join*, while a non-STRICT enrichment is achieved by means of a *left outer join*[4].

6.1 Schema Extension

SESQL provides three ways a schema can be extended. We discuss these next.

6.1.1 SCHEMAEXTENSION(attribute, property)

Intuitively, `SCHEMAEXTENSION(attribute, property)` is the clause that enables the user to add an attribute (not coming from the database schema) to the relation returned by the SQL part of the query, and the ontological properties based on which the values for the new attribute will be computed. Informally, the enrichment is obtained by (i) creating a SPARQL query to find the RDF triples containing the specified property `property`; (ii) comparing the values of the attribute `attribute`, an attribute occurring in the SELECT clause of the SQL query, with the subjects of the returned RDF triples. In case of match, the corresponding objects are returned as the values for the new column of the SESQL query result.

Example 1 (Schema Extension) Let us consider the SmartGround scenario reported in Section 3 and let us suppose that the user Alice is interested in knowing the content (in terms of elements) of a given landfill, 'nordLF', using the database visualized in Figure 1. Let us further assume that she also would like to complete this information by indicating for each element how dangerous it can be for the environment (if this information is available), according to the knowledge stored in the ontology (see Figure 2). She can specify this with the following SESQL query:

[4] In case of multiple enrichments by select, each resulting knowledge table is combined with the others by means of an additional join in the `FROM` clause of the query.

```
SELECT elem_name
  FROM elem_contained
  WHERE landfill_name = "nordLF"
ENRICH
  SCHEMAEXTENSION(elem_name, danger_level)
  KLEVEL(COMMON, PERSONAL)
```

In this case, the semantic enrichment process proceeds by (i) evaluating the `SCHEMAEXTENSION` clause in order to create a SPARQL query to find all the RDF statements in $KB(Alice)$ containing the specified property `danger_level` and (ii) comparing the values of the attribute `elem_name`, with the subjects of the returned statements. Based on the arguments for the `KLEVEL` clause, the set of statements involved in the query execution, $KB(Alice)$, will be selected. Given these, the system generates the following SPARQL query, which in turn gives the Knowledge Table KT (see Table 1(a)), listing the danger level of each element according to Alices' ontology (see Figure 2), where V = `elem_name` and V' = `danger_level`:

```
SELECT ?elem_name ?danger_level
WHERE{
  ?stm rdf:subject    ?elem_name .
  ?stm rdf:predicate  sg:danger_level .
  ?stm rdf:object     ?danger_level .
  ?user sg:userStatement ?stm .
  filter(regex(str(?user), "Alice") || regex(str(?user), "common"))}
```

Here, the first three predicates in the `WHERE` clause specify the statements involving the danger_level property, whereas the last predicate (and the associate filter) provides the scope of the query, as specified in the `KLEVEL` clause[5]. Next, the system executes the SQL part of the SESQL query, obtaining the Temporary Table TT (see Table 1(b)), which reports the element contained in landfill 'nordLF':

```
SELECT elem_name
  FROM elem_contained
  WHERE landfill_name = "nordLF"
```

Finally, the knowledge table KT and the temporary table TT (both stored in the Support DB) are joined, in order to obtain the enriched result:

```
SELECT TT.elem_name, danger_level
  FROM TT LEFT JOIN KT ON TT.elem_name = KT.elem_name
```

Table 1(c) presents the results; i.e., the content of the landfill *'nordLF'* also indicating, for each element, the relative danger level. In this example, Alice does not use the keyword "STRICT"; thus, for those data in the database for which the ontological knowledge regarding the dangerousness for the environment is not specified (e.g. cobalt), the `danger_level` attribute takes `null` value in the result. This is achieved through a *left join*. ◊

6.1.2 SCHEMAEXTENSION(attribute, property, concept)

In this special case, which we refer to as *Boolean schema extension*, the result of the initial SQL query is extended with a new column which can only assume Boolean values. Intuitively, given a relational attribute `attribute` (from the schema resulting from the SQL query, i.e., listed in the `SELECT` clause), an ontological property `property` and an ontological concept `concept`, for every value of `attribute` which is related to the given `concept` through the specified `property` in the ontological knowledge base, the value *true* will appear in the extension Boolean column, all the other values will be associated to the value *false*.

[5] For readability, we used human-readable, descriptive names to the variables in the SPARQL queries. In reality, in automatically generated queries, these names are not descriptive.

Table 2 Temporary tables and the result set for Example 2

elem_name	hazardous
mercury	true
cyanide	true

(a) KT

elem_name
mercury
cobalt

(b) TT

elem_name	hazardous
mercury	true
cobalt	false

(c) SESQL result

Example 2 (Boolean Schema Extension) Alice is interested in knowing whether the elements contained in the landfill 'nordLF' can be classified as hazardous or not, according to the database (Figure 1) and her knowledge base (Figure 2):

```
SELECT elem_name
  FROM elem_contained
  WHERE landfill_name = "nordLF"
ENRICH
  SCHEMAEXTENSION(elem_name, is_a, HazardousWaste)
  KLEVEL(COMMON, PERSONAL)
```

Similarly to Example 1, the semantic enrichment process proceeds by evaluating the `SCHEMAEXTENSION` clause. However, since an additional parameter (HazardousWaste) has been specified, a different SPARQL query is generated, this time returning a Boolean result. Namely, for each element in DB which is subject of an RDF statement in $KB(Alice)$ having property `is_a` and object `HazardousWaste`, the variable 'hazardous' will be set to "true" (see Table 2(a)):

```
SELECT ?elem_name ?hazardous
WHERE{
  ?stm rdf:subject    ?elem_name .
  ?stm rdf:predicate  sg:is_a .
  ?stm rdf:object     sg:hazardous_waste .
  ?user sg:userStatement ?stm .
  filter(regex(str(?user), "Alice") || regex(str(?user), "common"))
  BIND(true AS ?hazardous)}
```

Next, the system executes the SQL part of the SESQL query, obtaining the Temporary Table TT (see Table 2(b)), listing the content of the landfill 'nordLF'. Finally the knowledge table KT and the temporary table TT are joined:

```
SELECT elem_name,
  CASE
    WHEN hazardous IS NULL
    THEN false
    ELSE hazardous
  END
  FROM TT LEFT JOIN KT ON TT.elem_name = KT.elem_name
```

The semantically enriched result is presented in Table 2(c). The table lists the contents of the landfill *'nordLF'* and indicates for each element (in consequence of the non-STRICT policy) whether they are dangerous or not according to Alice's knowledge base. Note that, for the elements (e.g. cobalt) for which the knolwedge base does not provide any hazard related information (see Figure 2), the value in the newly added column is set to "false" (closed world assumption). ◊

6.1.3 SCHEMAEXTENSION(attribute_set, SPARQL_stmt, output_set)

This generalizes the schema extension process: here, `attribute_set` denotes the set of source attributes that will be used for matching the data results of the SQL query with the knowledge base; `SPARQL_stmt` denotes the user-provided SPARQL statement to be used for collecting the information from the knowledge base, and `output_set` denotes the set of attributes for schema extension. The results

Table 3 Temporary tables and result set for Example 3

elem_name
copper
iron

(a) TT

elem_name	extractionCost
copper	150
iron	100
zinc	150

(b) KT

elem_name	extractionCost
copper	150
iron	100

(c) SESQL result

of the SPARQL query are combined into a knowledge table KT with attributes `attribute_set` $\cup$ `output_set` and, as described earlier, this table is joined with the result table, $RES(A_1, \ldots, A_r)$, of the SQL query to obtain the result set $RES' = RES \bowtie_{A_i \in \texttt{attribute_set}} KT^{\texttt{SPARQL_stmt}}_{\texttt{attribute_set,output_set}}$.

Example 3 (Generalized Schema Extension) Alice needs the extraction costs for the elements contained in the landfill 'capitalLF'. Unlike the previous examples, Alice is interested in only the elements for which this additional information is made explicit in the ontology and, thus, adopts a STRICT policy:

```
SELECT elem_name
  FROM elem_contained
  WHERE landfill_name = "capitalLF"
ENRICH
  STRICT SCHEMAEXTENSION(elem_name, costQuery, extractionCost)
  DEFINE costQuery AS ${
 SELECT ?elem_name ?extractionCost
 WHERE {
  ?elem_name sg:extraction_method ?extractionMethod .
  ?extractionMethod sg:extraction_cost ?extractionCost}}$
```

First, the system executes the SQL part of the SESQL query, obtaining the temporary table TT which reports the content of the landfill 'capitalLF' (Table 3(a)):

```
SELECT elem_name
  FROM elem_contained
  WHERE landfill_name = "capitalLF"
```

Then, the user written SPARQL query is evaluated, resulting in the knowldge table KT, shown in Table 3(b), associating to each element the related extraction cost. As can be seen in Figure 2, this information is indirectly derived from the extraction method related to each specific element: "eddy_current_separator" for copper, with cost 150, and "magnetic_separator" for iron, with cost 100. Lastly, the system joins (this time by means of an *inner join*) the partial results,

```
SELECT elem_name, extractionCost
FROM TT JOIN KT ON TT.elem_name = KT.elem_name
```

providing the output listed in Table 3(c). In this table, the content of the landfill 'capitalLF', copper and iron, is associated to the relative cost of extraction. ◊

6.2 Schema Replacement

`SCHEMAREPLACEMENT` enables the users to replace one or more columns from the results of the SQL query with other columns, extracted from the knowledge base. SESQL provides three ways a schema can be replaced. We discuss these next.

Table 4 Temporary tables and result set for Example 4

elem_name	chemical_symbol
iron	Fe
zinc	Zn

(a) KT

elem_name
iron
zinc

(b) TT

chemical_symbol
Fe
Zn

(c) SESQL result

Table 5 Temporary tables and result set for Example 5

city	in_Piemonte
Campello Monti	true
Torino	true

(a) KT

landfill	city
capitalLF	Roma
alpLF	Torino
nordLF	Milano
littleLF	Campello Monti

(b) TT

landfill	in_Piemonte
capitalLF	false
alpLF	true
nordLF	false
littleLF	true

(c) SESQL result

6.2.1 SCHEMAREPLACEMENT(attribute, property)

This clause enables the users to replace columns from the results of the SQL query. The values of the replacing attributes are computed from the knowledge base.
Example 4 (Schema Replacement) Alice is interested in the content of a given landfill, 'littleLF', according to the database in Figure 1 and her knowledge base in Figure 2; but, instead of the name, she would like to have displayed the chemical symbol of each element be. She will pose the following SESQL query:

```
SELECT elem_name
  FROM elem_contained
  WHERE landfill_name = "littleLF"
ENRICH
  STRICT SCHEMAREPLACEMENT(elem_name, chemical_symbol)
  KLEVEL(COMMON, PERSONAL)
```

In this case, the system generates the following SPARQL query according to the `SCHEMAREPLACEMENT` clause parameters:

```
SELECT ?elem_name ?chemical_symbol
WHERE{
  ?stm rdf:subject    ?elem_name .
  ?stm rdf:predicate  sg:chemical_symbol .
  ?stm rdf:object     ?chemical_symbol .
  ?user sg:userStatement ?stm .
  filter(regex(str(?user), "Alice" ) || regex(str(?user), "common"))
}
```

The resulting knowledge table contains, for each element, the chemical symbol (Table 4(a)). Given this, the system executes the SQL part of the query, obtaining the temporary table TT (Table 4(b)), which lists the elements contained in the landfill 'littleLF'. Finally, the knowledge and the temporary tables are joined (again by means of an *inner join*), giving the results presented in Table 4(c); note that, unlike the results from the original database, which include the names of the elements, the enriched results are the form of chemical symbols. ◊

6.2.2 SCHEMAREPLACEMENT(attribute, property, concept)

This version of the `SCHEMAREPLACEMENT` clause introduces a Boolean attribute and replaces it over the attribute "`attribute`" appearing as a parameter of the clause.

Example 5 (Boolean Schema Replacement) Alice wants to know whether the landfills listed in the database are located in the 'Piemonte' region or not, exploiting the nearest city data, available in DB (Figure 1) and the region/city information in the knowledge base (Figure 2). She uses the following SESQL query:

```
SELECT landfill_name, city
  FROM landfill
ENRICH
  SCHEMAREPLACEMENT(city, region, Piemonte)
  KLEVEL(COMMON, PERSONAL)
```

The system generates a SPARQL query according to the `SCHEMAREPLACEMENT` clause parameters, similarly to Example 2.

```
SELECT ?city ?in_Piemonte
WHERE{
  ?stm rdf:subject    ?city .
  ?stm rdf:predicate  sg:region .
  ?stm rdf:object     sg:piemonte .
  ?user sg:userStatement ?stm .
  filter(regex(str(?user), "Alice") || regex(str(?user),  "common"))
  BIND(true AS ?in_Piemonte)}
```

The resulting knowledge table contains a true value for each city in the Piemonte region (see Table 5(a)). Next, the system executes the SQL part of the SESQL query, obtaining the temporary table TT (see Table 5(b)), reporting, for each landfill, its nearest city. Finally, the knowledge and the temporary tables are joined:

```
SELECT landfill_name,
  CASE
    WHEN in_Piemonte IS NULL
    THEN false
    ELSE in_Piemonte
  END
  FROM TT LEFT JOIN KT ON TT.city = KT.city
```

Results are presented in Table 5(c): as we see here, the semantically enriched table reports, for each landfill, whether it is situated in the Piemonte region or, not. ◊

6.2.3 SCHEMAREPLACEMENT(attribute_set, SPARQL_stmt, output_set)

This clause generalizes the schema replacement process with a user provided SPAQRL statement, as in Section 6.1.3. However, unlike *SCHEMAEXTENSION* discussed in that section, in this case, the original attributes in the `attribute_set` are removed from the result:

$$RES'' = \Pi_{(\{A_1,\ldots,A_r\}/\texttt{attribute_set})\cup\texttt{output_set}}(RES')$$

Example 6 (Generalized Schema Replacement) Alice is interested in the taxation (in percentage) imposed on each landfill by the corresponding region. Since, as can be seen in Figure 2, this information is indirectly associated to each landfill, Alice provides the SPARQL component of the SESQL query on her own:

```
SELECT landfill_name, city
  FROM landfill
ENRICH
  SCHEMAREPLACEMENT(city, taxQuery, tax)
  DEFINE taxQuery AS [
    SELECT ?city ?tax
    WHERE{
      ?city sg:region ?reg .
      ?reg sg:region_tax ?tax }]
```

Table 6 Temporary tables and the result set for Example 6

landfill_name	city
capitalLF	Roma
alpsLF	Torino
nordLF	Milano
littleLF	Campello Monti

(a) TT

city	tax
Roma	6
Torino	7
Campello Monti	7

(b) KT

landfill_name	tax
capitalLF	6
alpsLF	7
nordLF	null
littleLF	7

(c) SESQL result

Given this, the system first executes the SQL query obtaining the temporary table TT (Table 6(a)) which associates to each landfill its nearest city:

```
SELECT landfill_name, city
  FROM landfill
```

After evaluating the SPARQL query provided by the user, the system joins the resulting knowledge table KT (Table 6(b)) with the temporary table TT. The resulting table (Table 6(c)) provides for each landfill the regional tax, according to the location of the nearest city. Since this information may not be available for every city (e.g., 'Milano') and since the user has not specified STRICT enrichment, some values are `null` (e.g., 'nordLF'). ◊

7 Enrichment by WHERE Clause

As we introduced earlier in Section 5.1, enrichment process may also be applied to the `WHERE` clause *before* the condition evaluations. In particular, two types of enrichment by `WHERE` clause are possible:

- In *replacement of constants*, some of the constants in the user's SQL query are replaced by other (related) values obtained from the knowledge base.
- In *replacement of variables*, one or more of the attributes in the database are replaced by other (related) information extracted from the knowledge base.

Depending on whether they appear in *positive* or *negative* conditions, the replaced values contribute in the query processing by extending or restricting the domain of the involved variable: positive conditions (such as equality) are considered as satisfied (i.e. true) whenever they are satisfied by *at least one* of the replacement values, while negative conditions (such as non-equality) are considered as satisfied (i.e. true) whenever they are not satisfied by *any* of the replacement values. If the SESQL query follows a non-STRICT policy, the original condition is combined with the one produced by the enrichment. In case of STRICT policy, instead, only the condition resulting from the enrichment contributes to the result[6].

7.1 Replacement of Constants

SESQL provides two distinct ways constants in the query can be replaced:

- `REPLACECONSTANT(label, const, property)`
- `REPLACECONSTANT(label, const, SPARQL_stmt)`

Let us consider the following SESQL query:

[6] In case of multiple enrichments, each additional condition is combined in the `WHERE` clause of the rewritten query (see example 10).

```
SELECT ...
  FROM ...
  WHERE ... $label{θ(A, 'c')} ...
ENRICH
  REPLACECONSTANT(label, 'c', [property | Q])
```

The syntax $\$label\{\ldots\}$ is used to mark the predicate(s) which will be subject to replacement. In the above example, only one constant, 'c', in only one predicate, $\theta(A, \text{'}c\text{'})$, has been marked for replacement. In general, the user can specify multiple replacement conditions and mark more than one query predicate for constant replacement. `REPLACECONSTANT` clause is then used for specifying how the constant in the correspondingly labeled predicate will be replaced.

Let $KT_c(V)$ be the knowledge table from a SPARQL query, either automatically generated from the RDF property `property` or provided by the user in the form of query Q (note that, this knowledge table is constrained to have one single attribute containing the values to be used for constant replacement). Depending on the nature of the predicate θ, the condition in a non-STRICT scenario will be rewritten as follows:

- $A = \text{'}c\text{'}$: This condition will be rewritten as

  A IN ($\{c\}$ $\bigcup$ SELECT V FROM KT_c)

- $A \neq \text{'}c\text{'}$: This condition will be rewritten as

  A NOT IN ($\{c\}$ $\bigcup$ SELECT V FROM KT_c)

- $A < \text{'}c\text{'}$, $A \leq \text{'}c\text{'}$: These conditions will be rewritten, respectively, as

  A < MAX ($\{c\}$ $\bigcup$ SELECT V FROM KT_c)
  A <= MAX ($\{c\}$ $\bigcup$ SELECT V FROM KT_c)

- $A > \text{'}c\text{'}$, $A \geq \text{'}c\text{'}$: These conditions will be rewritten, respectively, as

  A > MIN ($\{c\}$ $\bigcup$ SELECT V FROM KT_c)
  A >= MIN ($\{c\}$ $\bigcup$ SELECT V FROM KT_c)

The STRICT counterpart of the rules above, is obtained by dropping the "$\{c\}$ $\bigcup$" part. We next provide several examples.

Example 7 (Replacement of Constant) Alice is interested in the landfills which contain nickel. But, instead of just relying on the data in the database, she also wants to consider those landfills that are likely to contain nickel as implied by the 'ore_assemblage' property, which associates to a given element, the other elements that are usually found in the type of rocks (and hence in the mining waste) from which that element is extracted. To combine these results, she thus adopts a non-STRICT policy. She can formulate this using the following SESQL query against the database in Figure 1 and her knowledge base in Figure 2.

```
SELECT landfill_name,
  FROM elem_contained
  WHERE $label1{elem_name = "nickel"}
ENRICH
  REPLACECONSTANT(label1, nickel, oreAssemblage)
  KLEVEL(COMMON, PERSONAL)
```

As discussed earlier, the syntax $\$label\{\ldots\}$ within the SQL query is used to mark the predicate(s) in the query which will be subject to replacement (in this example, the use provided label is *label1*). The semantic enrichment process starts by evaluating the `REPLACECONSTANT` clause in order to create a SPARQL query to find all the RDF statements in $KB(Alice)$ having 'nickel' as subject and `ore_assemblage` as property. To achieve this, the system produces the following SPARQL query, which generates the knowledge table KT presented in Table 7(a):

Table 7 Temporary table and the result set for Example 7

element
cobalt
copper

(a) KT

landfill_name
capitalLF
alpLF
nordLF

(b) SESQL result

landfill_name
alpLF

(c) Results without enrichment

```
SELECT ?element
WHERE{
  ?stm rdf:subject    sg:nickel .
  ?stm rdf:predicate  sg:ore_assemblage .
  ?stm rdf:object     ?element .
  ?user sg:userStatement  ?stm .
  filter(regex(str(?user), "Alice") || regex(str(?user),  "common"))
}
```

Given the resulting knowledge table, the `WHERE` clause in the SQL part of the query is then rewritten in order to involve both the original condition and the one resulting from the enrichment (under the non-STRICT policy), according to the rules detailed above (where the disjunction implements the set membership):

```
SELECT landfill_name
  FROM elem_contained
  WHERE elem_name = "nickel" OR elem_name IN(SELECT element FROM KT)
```

Table 7(b) presents the SESQL results under semantic enrichment, listing the name of the landfills where nickel can be found (directly or potentially, according to the *ore_assemblage* property). As a comparison, consider Table 7(c), which illustrates the result of the corresponding SQL query executed on the database without enrichment: as we can see, without semantic enrichment, the result is missing several landfills, namely the ones where nickel is not explicitly reported. ◊

Example 8 (Generalized Replacement of Constant) Alice is interested in the landfills which contain hazardous chemical compounds, according to the database in Figure 1 and her knowledge base in Figure 2:

```
SELECT landfill_name, chem_name
  FROM chem_contained
  WHERE $label1{chem_name = "hazardous"}
ENRICH
  STRICT REPLACECONSTANT(label1, hazardous, dangerQuery)
  KLEVEL(COMMON, PERSONAL)
  DEFINE dangerQuery AS [
     SELECT ?chem_name
     WHERE { ?chem_name sg:is_a sg:hazardous_waste }]
```

The `DEFINE` clause, here, allows the user to provide a SPARQL query to be used for constructing the knowledge table. This query is executed on the knowledge base specified by Alice in the `KLEVEL` clause. The resulting knowledge table, KT, is presented in Table 8(a), and lists the chemical compounds indicated as hazardous in Alice's ontology (see Figure 2). Finally, the condition labeled *label1* in the SQL query is rewritten according to the rules presented earlier:

```
SELECT landfill_name, chem_name
  FROM chem_contained
  WHERE chem_name IN(SELECT chem_name FROM KT)
```

The result of the SESQL query is presented in Table 8(b). This table is populated by retrieving the list of hazardous chemical compounds from Alice's knowledge base (as reported in the knowledge table) and by using this information to extract from the database the list of landfills in which those compounds are contained. ◊

Table 8 Temporary table and the result set for Example 8

chem_name
cyanide

(a) KT

landfill_name	chem_name
capitalLF	cyanide
nordLF	cyanide

(b) SESQL result

7.2 Replacement of Variables

The user can enrich the `WHERE` clause also by replacing attributes, rather than the constants in the query. Similarly to the replacement of constants, SESQL provides two distinct ways variables in the query can be replaced:

- `REPLACEVARIABLE(label, attribute, property)`
- `REPLACEVARIABLE(label, attribute, SPARQL_stmt)`

Let us consider the following SESQL query:

```
SELECT ...
  FROM ...
  WHERE ... $label{θ(A, B)} ...
ENRICH
  REPLACEVARIABLE(label, B, [p | Q])
```

This query is similar to the `REPLACECONSTANT` one, except that what is marked for replacement here is not a constant, but a query attribute. Consequently, the resulting knowledge table $KT_B(V, V')$, will have two attributes, V corresponding to the values to matched against the original table and V' corresponding to the replacement values from the knowledge base, obtained using a SPARQL query either automatically generated (thus involving the RDF property `property`), or written by the user (referred to as the query Q). Therefore, the rewrite semantics (under the non-STRICT enrichment policy) also needs to reflect this:

- $A = B$: This condition will be rewritten as

  A `IN (SELECT` V' `FROM` KT_B `WHERE` $V = B$`) OR` $A = B$

- $A \neq B$: This condition will be rewritten as

  A `NOT IN (SELECT` V' `FROM` KT_B `WHERE` $V = B$`) AND` $A \neq B$

- $A < B$, $A \leq B$: These conditions will be rewritten, respectively, as

  A `<  MAX (SELECT` V' `FROM` KT_B `WHERE` $V = B$`) OR` $A < B$
  A `<= MAX (SELECT` V' `FROM` KT_B `WHERE` $V = B$`) OR` $A \leq B$

- $A > B$, $A \geq B$: These conditions will be rewritten, respectively, as

  A `>  MIN (SELECT` V' `FROM` KT_B `WHERE` $V = B$`) OR` $A > B$
  A `>= MIN (SELECT` V' `FROM` KT_B `WHERE` $V = B$`) OR` $A \geq B$

The STRICT counterpart of the rules above, is obtained by dropping the last term of each rule.

Example 9 (Replacement of Variable) Alice is interested in pairs of landfills which share at least one element. As in earlier examples, she would like to leverage the *ore_assemblage* property to consider those elements that are not directly reported in the database. She can formulate the following SESQL query against the database in Figure 1 and her knowledge base in Figure 2:

Table 9 Temporary table and the result set for Example 9

waste1	waste2
nickel	cobalt
nickel	copper
zinc	barite
zinc	fluorite
sulfur	chlorine

(a) KT

land1	land2	elem
capitalLF	littleLF	iron
alpsLF	littleLF	zinc
alpsLF	nordLF	cobalt
alpsLF	capitalLF	copper
capitalLF	nordLF	chlorine

(b) SESQL result

land1	land2	elem
capitalLF	littleLF	iron
alpsLF	littleLF	zinc

(c) Results without enrichment

```
SELECT e1.landfill_name AS land1, e2.landfill_name AS land2,
 e1.elem_name AS elem
  FROM elem_contained AS e1, elem_contained AS e2
  WHERE $label1{e1.elem_name = e2.elem_name} AND e1.landfill_name <>
      e2.landfill_name
ENRICH
  REPLACEVARIABLE(label1, e2.elem_name, ore_assemblage)
```

In this case, the condition marked by the user has two variable names, corresponding to attributes from the two input relations. The `REPLACEVARIABLE` clause identifies that the variable, `e2.elem_name`, is to be replaced with information coming from the knowledge base. In particular, the replacement involves RDF statements in $KB(Alice)$ having `ore_assemblage` as the property. Thus, the system first produces the following SPARQL query, which generates the knowledge table KT (visualized in Table 9(a)) with two attributes, *waste1* and *waste2*, which are related in Alice's ontology by means of the *ore_assemblage* property.

```
SELECT ?waste1 ?waste2
WHERE{
  ?stm rdf:subject    ?waste1 .
  ?stm rdf:predicate  sg:ore_assemblage .
  ?stm rdf:object     ?waste2 .
  ?user sg:userStatement | sg:userInherit ?stm
  filter(regex(str(?user), "Alice" ) || regex(str(?user), "common"))
}
```

Given the resulting knowledge table, the labeled condition in the SQL is rewritten according to the rules detailed above:

```
SELECT e1.landfill_name AS land1, e2.landfill_name AS land2,
e1.elem_name AS elem
FROM elem_contained AS e1, elem_contained AS e2
WHERE (e1.elem_name = e2.elem_name
 OR e1.elem_name IN(
    SELECT waste2 FROM KT WHERE waste1 = e2.elem_name))
 AND e1.landfill_name <> e2.landfill_name
```

Table 9(b) presents the SESQL results obtained under variable replacement. The table lists pairs of landfills which share at least one element (either based on the data in the database or, potentially, according to the knowledge implied by the *ore_assemblage* property). As a comparison, Table 9(c) illustrates the result of the corresponding SQL query to the database without enrichment: as we can see, without semantic enrichment, the result contains only those pairs of landfills for which the shared elements are explicitly listed in the data tables. ◊

Note that, in the above example, we replaced the variable using a simple *property* based predicate. More complex user provided SPARQL statements can also be used (as in constant replacement), as specified earlier.

Example 10 (Generalized Replacement of Variable) Similarly to Example 9, Alice is interested in retrieving the landfills which share at least one element (again

Table 10 Temporary tables and result set for Example 10

waste1	waste2
nickel	cobalt
nickel	copper
zinc	barite
zinc	fluorite

(a) KT1

elem_name
zinc
nickel
iron
cobalt
copper
mercury

(b) KT2

land1	land2	elem
capitalLF	littleLF	iron
alpsLF	littleLF	zinc
alpsLF	nordLF	cobalt
alpsLF	capitalLF	copper

(c) SESQL Result

by also relying on the 'ore_assemblage' property), but this time restricting the results by considering only the elements that are classified as 'Metals', according to the common knowledge. In order to obtain these results, Alice has to define a twofold enrichment in the SESQL query. The first, *generalized replacement of variable*, deals with the part of the query regarding the *ore_assemblage* property, while the second, *Generalized replacement of constant*, addresses the limitation on the kind of elements desired (i.e., only metals). In both of these enrichments, Alice formulates the SPARQL part of the SESQL query on her own:

```
SELECT e1.landfill_name AS land1, e2.landfill_name AS land2,
    e1.elem_name AS elem
  FROM elem_contained AS e1, elem_contained AS e2
  WHERE $label1{e1.elem_name = e2.elem_name} AND e1.landfill_name <>
      e2.landfill_name AND $label2{e1.elem_name = "metal"}
ENRICH
  REPLACEVARIABLE(label1, e2.elem_name, oreQuery)
  DEFINE oreQuery AS [
    SELECT ?waste1 ?waste2
    WHERE{ ?waste1 sg:ore_assemblage ?waste2 .
           ?waste1 rdf:type sg:metal }]
  STRICT REPLACECONSTANT(label2, metal, metalQuery)
  DEFINE metalQuery AS [
    SELECT ?elem_name
    WHERE { ?elem_name sg:is_a sg:metal }]
```

Given this query, the system first executes the SPARQL subqueries, obtaining two knowledge tables: KT1 associates to each element those elements that might be present in the same waste product (see Table 10(a)); KT2 lists the elements classified as metals in the common ontology (see Table 10(b)). These knowledge tables are then leveraged to formulate the following enriched SQL query:

```
SELECT e1.landfill_name AS land1, e2.landfill_name AS land2,
  e1.elem_name AS elem
FROM elem_contained AS e1, elem_contained AS e2
WHERE (e1.elem_name = e2.elem_name OR e1.elem_name IN(
    SELECT waste2 FROM KT1 WHERE e2.elem_name = waste1))
  AND e1.elem_name IN(SELECT elem_name FROM KT2)
  AND e1.landfill_name <> e2.landfill_name
```

The result of the enriched query is reported in Table 10(c). The table lists pairs of landfills which share at least one metal (either based on the data in the database or according to the knowledge implied by the *ore_assemblage* property). When we compare this with the results presented in the previous example, we see that the pair of landfills that share "chlorine" has been dropped, since "chlorine" is not a metal. ◊

8 Semantically Enriched Query Processing Revisited

As we have seen earlier in Section 5.2 (*"Semantically Enriched Query Processing Overview"*), CroSSE integrates multiple data sources: (a) the *original database* contains pre-enrichment facts (organized in a relational database) and (b) the *ontology store* contains user provided enrichments (organized within an RDF store). A third database, referred to as the *support database (or SupportDB)* is used for stitching together the various data components to be presented to the end user. While, in general, the ontology store and the support database are physically co-located as part of the CroSSE software platform, the original database tends to be physically and logically (in terms of data model and access interface) separated from the other two. Consequently, the amount of data transferred from the original database to the semantic query module, SQM, and stored in the support database is one of the key performance bottlenecks for the system.

Consider, for instance, the enrichment by `SELECT` clause under STRICT policy (Section 6): in this case an SQL query is sent to the original database to obtain relevant tuples, which are then stored in a temporary table, TT, in the support database. TT is then joined with a knowledge table, KT, which stores the relevant knowledge extracted from the RDF store. We note, in this section, that the amount of data fetched from the original database can be significantly reduced if the SQL query to the original database is expanded to include, as a filter condition, the values of the join attribute in the KT table. This is analogous to implementing a semi-join and limits the tuples that need to be exchanged to only those that will be useful when combining TT and KT in the last phase. As we see in Table 11, sample queries Q1 through Q6 discussed in this paper benefit from this rewriting.

Optimization opportunities are not only limited to the enrichment by `SELECT` clause. In fact, the amount of data that needs to be transferred is especially high when implementing enrichment of `WHERE` requests: As we have seen in Section 7.1, for example, when replacing constants, we need to execute a query of type

```
SELECT A_i FROM ET
WHERE A_i IN(SELECT V FROM KT)
```

where the external table ET is available in the original database, whereas the knowledge table KT is located locally in the support database. A naive execution strategy would first pull ET in its entirety to the support database in the form of a temporary table, TT, and execute the above query locally. This, however, will likely require significant data transfer from the original database to the support database. Here, we note that this can be avoided by rewriting the query sent to the original database as

```
SELECT A_i FROM ET
WHERE A_i IN(k_1, ..., k_n)
```

where $k_1, \ldots, k_n$ are the results of the inner `SELECT` clause – which is nothing but the list of semantic annotations extracted from the RDF store using a SPARQL query. Note that this formulation not only reduces the amount of data fetched from the original database, but also eliminates the need to create a knowledge table in the support database. As we see in Table 11, sample queries Q7 and Q8 discussed earlier in this paper can benefit from this rewriting strategy.

Unfortunately, this strategy does not work when replacing variables. This is because, as we have seen in Section 7.2, variable replacement requires a nested query such that the filter condition of the inner `SELECT` clause requires data from

Table 11 Query rewriting

Query number	STRICT Policy (not optimized)	STRICT Policy (optimized)
Q1 to Q6	`TT = SELECT` A_i `FROM ET`	`TT = SELECT` A_i `FROM ET WHERE` A_i `IN(`$k_1, \ldots, k_n$`)`
Q7,Q8	`SELECT` A_i `FROM ET WHERE` A_i `IN(SELECT V FROM KT)`	`SELECT` A_i `FROM ET WHERE` A_i `IN(`$k_1, \ldots, k_n$`)`

both the locally-available knowledge table, KT, and an external table, ET, available at the original database. Therefore, executing this query requires pulling the relevant external data from the original database and populating a local temporary table, TT, on which such a query can be executed. We note that, even in this case, we can reduce the amount of data fetched from the original database by requesting only the data entries that will join with the knowledge table, KT, using a semi-join style filtering step. For example, to implement Q9 in Section 7.2, instead of fetching the complete `elem_contained` table from the original database, we can create a temporary table, TT, by executing the following query:

```
TT = SELECT elem_name, landfill_name
     FROM elem_contained
     WHERE elem_name IN("nickel","zinc", "sulfur") OR
           elem_name IN("cobalt", "copper", "barite", "fluorite", "
    chlorine")
```

This query returns all the relevant tuples because the only tuples in the `elem_contained` table that are relevant to the query are the ones that match the `waste1` or `waste2` attributes of the knowledge table, KT, reported in Table 9(a).

9 Conclusions

In this paper, we introduced *Crowd Sourced Semantic Enrichment (CroSSE)*, a crowdsourced knowledge platform supporting semantic enrichment and context-aware data access for scientific investigations. The semantic tagging module provides a set of functionalities that implement the belief-based knowledge expansion, allowing each user the possibility to (a) explore the common meta-knowledge, which is shared among all users; (b) extend common knowledge according to her domain of expertise (in particular by means of RDF statements connecting existing concepts through suggested properties and/or by defining new concepts and new properties); and (c) borrow (part of) the knowledge inserted by other users, possibly leading to an enrichment of the common knowledge. The SESQL language allows users to enrich a relational databank with semantic tagging information and poses contextualised queries to support contextualised data analysis. Many of the key functionalities of CroSSE have been deployed in the domain of tracking secondary raw materials in the context of the SmartGround project.

Acknowledgements The research is partially supported by EU grants #641988 and #690817 and NSF grant #1633381. We thank project partners, especially our colleagues from the Earth Sciences Department at the University of Torino, P. Rossetti, G. Dino, and G. Biglia.

References

1. Minc: A social platform for fostering educational interactions, 2017.

2. S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query processing in the sims information mediator. In *SIGMOD*, pages 137–148, 1996.
3. Y. Arens, C. Knoblock, and C. Hsu. Query processing in the sims information mediator. In *AAAI*, 1996.
4. D. Beckett, editor. *RDF/XML Syntax Specification (Revised).* W3C Recommendation. 2004.
5. E. G. Caldarola, A. Picariello, and A. M. Rinaldi. An approach to ontology integration for ontology reuse in knowledge based digital ecosystems. In *MEDES*, 2015.
6. A. Calì, G. Gottlob, and A. Pieris. Advanced processing for ontological queries. *Proc. VLDB Endow.*, 3(1-2):554–565, Sept. 2010.
7. K. S. Candan, H. Cao, Y. Qi, and M. L. Sapino. System support for exploration and expert feedback in resolving conflicts during integration of metadata. *VLDB Journal*, 17(6):22–119, 2008.
8. G. Cavallo, F. Di Mauro, P. Pasteris, M. L. Sapino, and K. S. Candan. Contextually-enriched querying of integrated data sources. In *ICDE18 Workshops*, 2018.
9. H. Davulcu, J. Freire, M. Kifer, and I. Ramakrishnan. A layered architecture for querying dynamic web content. In *SIGMOD*, 1999.
10. H. Davulcu, M. Kifer, G. Yang, and I. Ramakrishnan. Design and implementation of the physical layer in webbases: The xrover experience. In *DOOD*, 2000.
11. F. Di Mauro, P. Pasteris, M. L. Sapino, K. S. Candan, G. A. Dino, and P. Rossetti. Crowdsourced semantic enrichment for participatory e-government. In *Proceedings of the 8th International Conference on Management of Digital EcoSystems, MEDES 2016.*
12. F. Di Pinto, D. Lembo, M. Lenzerini, R. Mancinu, A. Poggi, R. Rosati, M. Riccardo, Ruzzi, and D. Savo. Optimizing query rewriting in ontology-based data access. In *EDBT*, 2013.
13. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajararnan, Y. Sagiv, J. Ullman, V.Vassalos, and J. Widom. The tsimmis approach to mediation: Data models and languages. *JIIS*, 2, 1997.
14. G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization. *2011 IEEE 27th International Conference on Data Engineering*, pages 2–13, 2011.
15. E. Kandogan, M. Roth, P. M. Schwarz, J. Hui, I. G. Terrizzano, C. Christodoulakis, and R. J. Miller. Labbook: Metadata-driven social collaborative data analysis. In *Int. Conf on Big Data 2015.*
16. A. Levy. The information manifold approach to data integration. *IEEE Intelligent Systems*, pages 1312–16, 1998.
17. L. Lim, H. Wang, and M. Wang. Semantic queries by example. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT 2013)*, 2013.
18. A. Maedche, S. Staab, R. Studer, Y. Sure, and R. Volz. Seal – tying up information integration and web site management by ontologies. *IEEE Data Engin. Bulletin*, 2002.
19. J. Mortensen, P. R. A. andM. A. Musen, and N. F. Noy. Crowdsourcing ontology verification. In *ICBO*, pages 40–45, 2013.
20. J. Mortensen, M. Musen, and N. Noy. Developing crowdsourced ontology engineering tasks: An iterative process. In *CrowdSem*, pages 79–88, 2013.
21. K. Munir and M. S. Anjum. The use of ontologies for effective knowledge modelling and information retrieval. *Applied Computing and Informatics*, 2017.
22. K. Munir, M. Odeh, and R. Mcclatchey. Ontology-driven relational query formulation using the semantic and assertional capabilities of OWL-DL. *Know.-Based Syst.*, 35:144–159, Nov. 2012.
23. S.Kambhampati, E. Lambrecht, U. Nambiar, Z. Nie, and G. Senthil. Optimizing recursive information gathering plans in emerac. *JIIS*, pages 22–119, 2004.
24. M. C. Suárez-Figueroa, A. Gómez-Pérez, E. Motta, and A. Gangemi. *Ontology Engineering in a Networked World*, chapter 2. Springer, 2012.
25. N. E. Taylor and Z. G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, 2006.
26. J. Tekli, R. Chbeir, A. J. Traina, C. Traina, K. Yetongnon, C. R. Ibanez, M. A. Assad, and C. Kallas. Full-fledged semantic indexing and querying model designed for seamless integration in legacy rdbms. *Data and Knowledge Engineering*, 117, 2018.
27. G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev. Ontology-based data access: A survey. In *IJCAI-18.*
28. L. Xu and D. W. Embley. Combining the best of global-as-view and local-as-view for data integration. In *ISTA*, pages 123–136, 2002.
29. P. Ziegler and K. R. Dittrich. *Data Integration - Problems, Approaches, and Perspectives*, chapter 3. 2007.