Ontology Evolution in Data Integration[1]

Haridimos Kondylakis, Dimitris Plexousakis, Yannis Tzitzikas

Information Systems Lab., FORTH-ICS, Computer Science Department, Univ. Of Crete
N.Plastira 100, Vassilika Vouton, GR70013, Heraklion, Crete, Greece
{kondylak, dp, tzitzik}@ics.forth.gr

Abstract. Ontologies are becoming more and more important in data integration. An important problem when dealing with ontologies is the fact that they are living artefacts and subject to change. When ontologies evolve, the changes should somehow be rendered and used by the pre-existing data integration systems. In most of these systems, when ontologies change their relations with the data sources i.e. the mappings, are recreated manually, a process which is known to be error-prone and time-consuming. In this paper, we provide a solution that allows query answering under evolving ontologies without mapping redefinition. In order to achieve that, we present a module that enables ontology evolution over traditional ontology-based data integration systems. This module gets as input the different ontology versions and the user query, and answers queries over data integration systems that use different ontology versions. We identify the problems in such a setting and we provide efficient, intuitive solutions. We prove that our approach imposes only a small overhead over traditional query rewriting algorithms and it is modular and scalable. Finally, we show that it can greatly reduce human effort spent since continuous mapping redefinition on evolving ontologies is no longer necessary.

1 Introduction

The development of new scientific techniques and the emergence of new high throughput tools have led to a new information revolution. During this information revolution the data gathering capabilities have greatly surpassed the data analysis techniques, making the task to fully analyze the data at the speed at which it is collected a challenge. The amount, diversity, and heterogeneity of that information have led to the adoption of data integration systems in order to manage it and further process it. However, the integration of these data sources raises several semantic heterogeneity problems.

By accepting an ontology as a point of common reference, naming conflicts are eliminated and semantic conflicts are reduced. During the last years, ontologies have been used in database integration [1], obtaining promising results, for example in the fields of biomedicine and bioinformatics. When using ontologies to integrate data, one is required to produce mappings, to link similar concepts or relationships from the ontology/ies to the sources (or other ontologies) by way of an equivalence - according

[1] This work was partially supported by the EU project PlugIT (ICT-231430)

to some metric. This is the *mapping definition process* [2] and the output of this task is the *mapping*, i.e., a collection of mappings rules. In practice, this process is done manually with the help of graphical user interfaces and it is a *time-consuming*, *labour-intensive* and *error-prone* activity. Defining the mappings between schemata/ontologies is not a goal in itself. The resulting mappings are used for various integration tasks such as data transformation and query answering.

Despite the great amount of work done in ontology-based data integration, an important problem that most of the systems tend to ignore is that ontologies are living artifacts and subject to change [3]. Due to the rapid development of research, ontologies are frequently changed to depict the new knowledge that is acquired. The problem that occurs is the following: when ontologies change, the mappings may become invalid and should somehow be updated or adapted.

In this paper, we address the problem of data integration for evolving RDF/S ontologies. We argue that ontology change should be considered when designing ontology-based data integration systems. A typical solution would be to regenerate the mappings and then regenerate the dependent artifacts each time the ontology evolves. However, as this evolution might happen too often, the overhead of redefining the mappings each time is significant. The approach, to recreate mappings from scratch each time the ontology evolves, is widely recognized to be problematic [4-6], and instead, previously captured information should be reused. However, all current approaches that try to do that suffer from several drawbacks and are inefficient [7] in handling ontology evolution in a state of the art ontology-based data integration system.

The lack of an ideal approach leads us to propose a new mechanism that builds on the latest theoretical advances on the areas of ontology change [8] and query rewriting [1, 9] and incorporates and handles ontology evolution efficiently and effectively. More specifically:

- We present the architecture of a data integration system, named *Evolving Data Integration (EDI)* system, that allows the evolution of the ontology used as global schema.
- We define the exact semantics of our system and we elegantly separate the semantics of query rewriting for different ontology versions and for the sources. Since query rewriting for the sources has been extensively studied [1, 2, 9], we focus on a layer above and deal only with the query rewriting between ontology versions. More specifically, we present a *module* that receives a user query specified under the latest ontology version and produces rewritings that will be answered by the underlying data integration systems - that might use different ontology versions. The query processing in this module consists of two steps: a) *query expansion* that considers constraints coming from the ontology, and b) *valid query rewriting* that uses the changes between two ontology versions to produce rewritings among them.
- In order to identify the changes between the ontology versions we adopt a high-level language of changes which possesses salient properties such as *uniqueness*, *inversibility* and *composability.* The sequence of changes between the latest and the other ontology versions is produced automatically at setup time and then each one of the change operations identified is translated into a logical GAV mapping. This translation enables query rewriting by unfolding. Then, the inversibility is

exploited to rewrite queries from past ontology versions to the current, and vice versa, and composability to avoid the reconstruction of all sequences of changes among the latest and all previous ontology versions.

- Despite the fact that query rewriting always terminates the queries issued to other ontology versions might fail. We show that this problem is not inhibiting in our algorithms but a consequence of information unavailability. To tackle this problem, we propose two solutions: either to provide more general answers, or to provide insights for the failure, thus driving query redefinition only for a specific portion of the affected query.

Such a mechanism, that provides rewritings among data integration systems that use different ontology versions, is *flexible*, *modular* and *scalable*. It can be used on top of any data integration system – independently of the family of the mappings they use (GAV, LAV, GLAV, etc [2]). New mappings or ontology versions can be easily and independently introduced without affecting other mappings or other ontology versions. Our engine takes the responsibility of assembling a coherent view of the world out of each specific setting.

The rest of the paper is organized as follows: Section 2 introduces the problem by an example and presents related work. Section 3 presents the architecture of our system and describes its components. Section 4 describes the semantics of our system and Section 5 elaborates on the query rewriting among ontology versions. Then, Section 6 presents the problems that may occur in such a setting and proposes elegant solutions. Finally, Section 7 provides a summary and an outlook for further research.

2 Motivating Example & Related Work

Consider the example RDF/S ontology shown on the left of Fig. 1. This ontology is used as a point of common reference, describing persons and their contact points. We also have two relational databases DB1 and DB2 mapped to that version of the ontology. Assume now that the ontology designer decides to move the domain of the "*has_cont_point*" property from the class "*Actor*" to the class "*Person*", and to delete the literal "*gender*". Moreover, the "*street*" and the "*city*" properties are merged to the "*address*" property as shown on the right of Fig. 1. Then, DB3 is mapped to the new version of the ontology leading to two data integration systems that work independently. In such a setting we would like to issue queries formulated using any ontology version available. Moreover, we would like to retrieve answers from all underlying databases.

Several approaches have been proposed so far to tackle similar problems. In a data exchange setting mapping adaptation [4] was one of the first attempts that tried to incrementally adapt the mapping between the schema and the data sources as the schema evolved. The idea was that schemata evolve in small primitive steps; after each step the schema mappings can be incrementally adapted by applying local modifications. However, several problems of the approach were early identified [5] (such as the multiple list of changes with the same effect e.t.c), and another approach was proposed [5]. The approach was to describe ontology evolution as mappings and to employ mapping composition to derive the adapted mappings. However, mapping composition proved to be a really difficult problem [10] and we have not yet seen an elegant approach for ontology evolution.

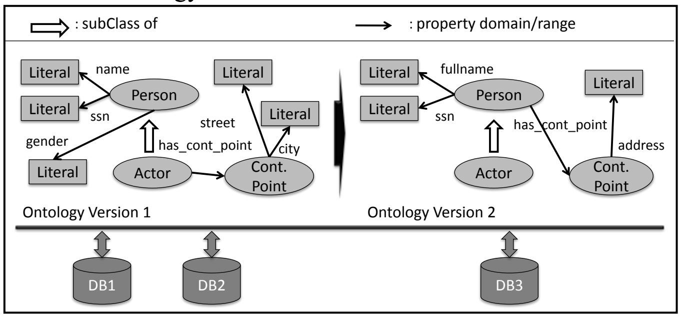

Fig. 1. The motivating example of an evolving ontology

Furthermore, several research works tried to deal with similar problems. For XML databases, for example, there have been several approaches that try to preserve mapping information under changes [11] or propose guidelines for XML schema evolution in order to maintain the mapping information [12]. Moreover, augmented schemata were introduced in [13] to enable query answering over multiple schemata in a data warehouse, whereas other approaches change the underlying database systems to store versioning and temporal information such as [14-17]. However, our system differs from all the above in terms of both goals and techniques. To the best of our knowledge no system today is capable of retrieving information mapped with different ontology versions.

3 Evolving Data Integration

We conceive an *Evolving Data Integration* (EDI) system as a collection of data integration systems, each using a different ontology version as global schema. Therefore, we extend the traditional formalism from [2] and define an EDI as:

Definition 3.1 (Evolving Data Integration System). *An EDI system I is a tuple of the form* $((O_1, S_1, M_1), \ldots, (O_m, S_m, M_m))$ *where* O_i *is a version of the ontology,* S_i *is a set of local sources and* M_i *is the mapping between* S_i *and* O_i $(1 \leq i \leq m)$.

Considering O_i we restrict ourselves to *RDF/S knowledge bases*, as most of the Semantic Web Schemas (85,45%) are expressed in RDF/S [18]. The representation of knowledge in RDF is based on triples of the form *predicate (subject, object)*. Assuming two disjoint and infinite sets U, L, denoting the URIs and literals respectively, $T = U \times U \times (U \cup L)$ is the set of all triples. An RDF Graph V is defined as a set of triples, i.e., $V \subseteq T$. RDFS [19] introduces some built-in classes (class, property) which are used to determine the *type* of each resource. The typing mechanism allows us to concentrate on nodes of RDF graphs, rather than triples, which is closer to ontology curators' perception and useful for defining intuitive high-level changes. RDFS provides also *inference semantics*, which is of two types, namely *structural inference* (provided mainly by the transitivity of subsumption relations) and *type inference* (provided by the typing system, e.g., if p is a property, the triple (p, *type*, *property*) can be inferred). The RDF Graph containing all triples that are either explicit or can be inferred from explicit triples in an RDF Graph V (using *both* types of inference), is called the *closure* of V and is denoted by $Cl(V)$. An *RDF/S Knowledge Base (RDF/S KB)* V is an RDF Graph which is closed with respect to *type inference*, i.e., it contains all the triples that can be inferred from V using type inference.

Moreover, we consider relational databases as source schemata. We choose to use relational databases since the majority of information currently available is still stored on relational databases [20].

For modelling ontology evolution we use a high-level language of changes that describes how an ontology version was derived from another ontology version. A high-level language is preferable than a low-level one, as it is more *intuitive*, *concise*, *closer to the intentions* of the ontology editors and *captures more accurately* the semantics of change [8]. As we shall see later on, a high-level language is beneficial for our problem for two reasons: First, because the produced change log has a smaller size and second, because such a language yields logs that contain a smaller number of individual low-level deletions (which are non-information preserving) and this affects the effectiveness of our rewriting. Moreover properties like *composability* and *inversibility* can be exploited for improving efficiency as we shall see on the sequel. In our work, a change operation is defined as follows:

Definition 3.2 (Change Operation). *A change operation u over O, is any tuple* (δ_a, δ_d) *where* $\delta_a \cap O = \emptyset$ *and* $\delta_d \subseteq O$. *A change operation u from* O_1 *to* O_2 *is a change operation over* O_1 *such that* $\delta_a \subseteq O_2 \setminus O_1$ *and* $\delta_d \subseteq O_1 \setminus O_2$.

Obviously, δ_a and δ_d are sets of triples end especially the triples in δ_d are triples coming from the ontology O. For simplicity we will denote $\delta_a(u)$ $(\delta_d(u))$ the *added (deleted)* triples of a change u. From the definition, it follows that $\delta_a(u) \cap \delta_d(u) = \emptyset$ and $\delta_a(u) \cup \delta_d(u) \neq \emptyset$ if $O_1 \neq O_2$. For the change operations proposed in [8] and the corresponding detection algorithm, it has been proved that the sequence of changes between two ontology versions is *unique.* Moreover, it is shown that for any two changes u_1, u_2 in such a sequence it holds that $\delta_a(u_1) \cap \delta_a(u_2) = \emptyset$ and $\delta_d(u_1) \cap \delta_d(u_2) = \emptyset$. These nice properties and their consequences are among the reasons that led us to adopt that specific language for describing changes among ontologies. Hereafter, whenever we refer to a change operation, we mean a change operation from those proposed in [8]. Now we need to define their application semantics.

Definition 3.3 (Application semantics of a high-level change). *The application of a change u over O, denoted by u(O), is defined as:* $u(O) = (O \cup \delta_a(u)) \setminus \delta_d(u)$.

Two key observations here are that the application of out change operations is not conditioned by the current state of the ontology and that we don't handle inconsistency, i.e., $(O \cup \delta_a(u)) \setminus \delta_d(u)$ is always assumed to be consistent).In our example the change log between O_2 and O_1, denoted by the E^{O_2,O_1}, consists of the following change operations:

***u_1**:Rename_Property(fullname, name)* ***u_2**:Split_Property(address, {street, city})*
***u_3**:Specialize_Domain(has_cont_point, Person, Actor)*
***u_4**:Add_Property(gender, ø, ø ,ø ,ø, Person, xsd:String, ø, ø)*

The definition of the change operations that are used in this paper can be found on [8]. It is obvious, that applying those change operations on O_2, results O_1. Now it is time to define the composition of the change operations. By proving that the change operations are composable, we will be able to use the intermediate evolution logs between ontology versions instead of constructing all change logs between the latest ontology version and all past ontology versions.

Definition 3.4 (Composition of change operations). *A u_{comp} is the composition of u_1 and u_2 (computed over O_1 and O_2), if $u_{comp}(O_1) = u_2(u_1(O_1)) = u_1(u_2(O_1))$*

Now we will show that the change operations as detected in [8] compose indeed.
Proposition 1: *Let u_1, u_2 two change operations from O_1 to O_2. Then $u_{comp} = (\delta_a(u_1) \cup \delta_a(u_2), \delta_d(u_1) \cup \delta_d(u_2))$*

Finally, since a change operation is actually a mapping function that maps O_1 to O_2, a question is whether there exists the *inverse* function, the inverse change operation that maps the O_2 to the O_1 ontology version. By automatically constructing the inverse of a sequence of change operations (from O_1 to O_2), we will be able to rewrite queries expressed using O_2 to O_1 and vice versa.

Definition 3.5 (Inverse of a change operation). *Let u be a change operation from O_1 to O_2. A change operation u_{inv} from O_2 to O_1 to is the inverse of u if:* $u_{inv}(u(O_1)) \equiv O_1$

Now we will show how to compute the inverse of a change operation.

Proposition 2: *The inverse of a change operation u (denoted by inv(u)) from O_1 to O_2 is: $inv(u)=(\delta_d(u), \delta_a(u))$*

Based on Propositions 1 and 2 we can conclude that:

Corollary 1: *The inverse of a sequence of change operations $E^{O_1,O_2} = [u_1, ..., u_n]$ constructed from O_1 to O_2, is $E_{inv}^{O_1,O_2} = [inv(u_n), ..., inv(u_1)]$.*

The inverse of the sequence of change operations for our running example is:
$inv(u_4)$:Delete_Property(gender, ø, ø ,ø ,ø, Person, xsd:String, ø, ø)
$inv(u_3)$:Generalize_Domain(has_cont_point, Actor, Person)
$inv(u_2)$:Merge_Properties({street, city},address)
$inv(u_1)$:Rename_Property(name, fullname)

4 Semantics of an EDI

Now we will define semantics for an EDI system *I*. **Fig. 2** sketches the proposed approach. We start by considering a *local database* for each (O_i, S_i, M_i), i.e., a database D_i that conforms to the local sources of S_i. Based on D_i, we shall specify which is the information content of the global schema O_i.

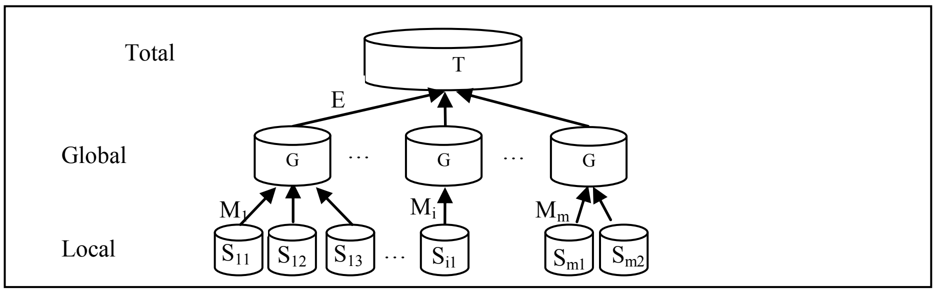

Fig. 2. The semantics of an EDI

Definition 4.1 (Legal global database): *A global database G_i for (O_i, S_i, M_i) is said to be legal with respect to D_i, if: (a) G_i is legal with respect to O_i, i.e., G_i satisfies all the constraints of O_i and (b) G_i satisfies the mapping M_i with respect to D_i.*

The notion of G_i satisfying the mapping M_i, with respect to D_i, is defined as it is commonly done in traditional data integration systems (see [2] for more details). It depends on the different assumptions that can be adopted for interpreting the tuples that D assigns to relations in local sources with respect to tuples that actually satisfy (O_i, S_i, M_i). Since such systems have been extensively studied in the literature we abstract from the internal details and focus on the fact that for each (O_i, Si, M_i) of our system we can obtain a global database G_i.

Now, we can repeat the same process, i.e., to consider the global databases as sources and a database $\mathfrak{D}$ which we will simply call the *global database*, the database that conforms to them. Now we can define the *legal total database.* We use the term "*total*" only to differentiate it from a *global* database, since we will extensively use it from now on.

Definition 4.2 (Legal total database): *A total database T for and EDI I is said to be legal with respect to $\mathfrak{D}$, if: (a) T is legal with respect to O_m, i.e., T satisfies all the constraints of the latest ontology version O_m. (b) T satisfies E with respect to $\mathfrak{D}$ ($E = \bigcup_1^{m-1} E^{O_m, O_i}$).*

The constraints of an RDF/S ontology are the inclusion dependencies among the classes and the properties. Now we specify the notion of T satisfying E with respect to $\mathfrak{D}$. In order to exploit the strength of the logical languages towards query reformulation, we convert our change operations to GAV mappings. So when we refer to the notion of T satisfying E we mean T satisfying the GAV mappings produced from E. A GAV mapping associates to each element g in T a query q_G over $G_1, \dots, G_m$, i.e., $q_G \rightarrow g$.

Definition 4.3 *A database T satisfies the mappings $q_G \rightarrow g$ with respect to $\mathfrak{D}$ if $q_G^{\mathfrak{D}} \subseteq g^T$ where $q_G^{\mathfrak{D}}$ is the result of evaluating the query q_G over $\mathfrak{D}$.*

For example, the sequence of the GAV mappings that corresponds to our sequence of changes is:

mu_1: $\forall fullname$, *type(fullname, property)* $\rightarrow \exists$*name, type(name , property)*

mu_2: $\forall$ *address, type(address, property)* $\rightarrow \exists$*street, city, type(street, property)* $\wedge$ *type(city, property)*

mu_3: $\forall Person$, *domain(has_cont_point , Person)* $\rightarrow$ $\exists Actor$, *domain(has_cont_point, Actor)*

For u_4 there is no GAV mapping constructed since we do not know where to map the deleted element. Now it becomes obvious that the lower the level of the language of changes used the more change operations won't have corresponding GAV mappings (since more low-level individual additions and deletions will appear).

By the careful separation between *the legal total database T* and *the legal global databases* G_i we have achieved the modular design of our EDI system and the separation between the traditional data integration semantics and the additions we have imposed in order to enable ontology evolution. Thus, our approach can be applied on top of any existing data integration system to enable ontology evolution.

5 Query Processing

Queries to I are posed in terms of the global schema O_m. For querying, we adopt the language SPARQL [21]. We chose SPARQL since it is currently the standard query language for the semantic web and has become an official W3C recommendation. Essentially, SPARQL is a graph-matching language. Given a data source, a query consists of a pattern (the graph pattern) which is matched against, and the values obtained from this matching are processed to give the answer. A SPARQL query consists of three parts: The *pattern matching part*, which includes several features of pattern matching of graphs, the *solution modifiers*, which once the output of the pattern has been computed (in the form of a table of values of variables), allows to modify these values applying classical operators, and the *output* of a SPARQL query which can be of different types: yes/no answers, selections of values etc. In order to avoid ambiguities in parsing, we present the syntax of SPARQL graph patterns in a more traditional algebraic way, using the binary operators *UNION (*denoted by **U***) AND* and *OPT*, and *FILTER* according to [21]. In this paper, we do not consider *OPT* and *FILTER* operators since we leave it for future work. The remaining SPARQL fragment we consider here corresponds to union of conjunctive queries [21]. Moreover, the application of the *solution modifiers* and the *output is* done after the evaluation of the query, and is not presented here.

Continuing our example, assume that we would like to know the "*ssn*" and "*fullname*" of all persons stored on our DBs and their corresponding address. The SPARQL query, formulated using the latter version of our example ontology is:

```
q1: select ?SSN ?NAME ?ADDRESS where {
?X type Person.               ?X ssn ?SSN.           ?X fullname ?NAME.
?X has_contact_point ?Y.      ?Y type Cont.Point.    ?Y address ?ADDRESS}
```

Using the semantics from [21] the algebraic representation of q_1 is equivalent to:

q_1: $\pi_{?SSN,?NAME,?ADDRESS}$ *((?X, type, Person) AND (?X, ssn, ?SSN) AND (?X, fullname, ?NAME) AND (?X, has_contact_point, ?Y) AND (?Y, type, Cont.Point) AND (?Y, address, ?ADDRESS))*

Now we define what constitutes an answer to a query over O_m. We will adopt the notion of *certain answers* [2, 9].

Definition 5.1 (Certain answers): *Given a global database $\mathfrak{D}$ for I, the answer $q^{I,\mathfrak{D}}$ to a query q with respect to I and $\mathfrak{D}$, is the set of tuples t such that $t \in q^T$ for every total database T that is legal for I with respect to $\mathfrak{D}$, i.e. such that t is an answer to q over every database T that is legal for I with respect to $\mathfrak{D}$. The set $q^{I,\mathfrak{D}}$ is called the set of certain answers to q with respect to I and $\mathfrak{D}$.*

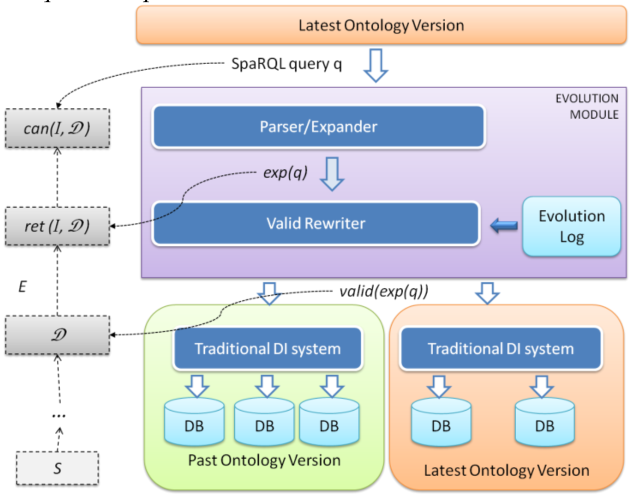

Fig. 3. Query processing

Note that, from a logical point of view, finding certain answers is a logical implication problem: check whether it logically follows from the information in the global databases G_i that t satisfies the query. It has been shown [22], that computing certain answers to union of conjunctive queries over a total database with constraints, corresponds to evaluating the query over a special database called *canonical* which represents all possible total databases legal for the data integration system and which may be infinite in general. However, instead of trying to construct the canonical database and then evaluate the query, another approach is to transform the original query q into a new query $exp_{O_m}(q)$ over the O_m, (which is called the *expansion* of *q* w.r.t. O_m) such that the answer to $exp_{O_m}(q)$ over the *retrieved total database* is equal to the answer to *q* over the *canonical database* [22]. We have to note however, that this approach holds for inclusion dependencies but not for the more general class of FOL constraints.

Definition 5.2 (retrieved total database): *If $\mathfrak{D}$ is a global database for the EDI-system I, then the retrieved total database ret(I, $\mathfrak{D}$) is the total database obtained by computing and evaluating, for every element of O_m the query associated to it by E over the global database $\mathfrak{D}$.*

This is a common approach in data integration under constraints, and we also adopt it here. This step is performed by the "*Parser/Expander*" component shown on Fig. 3. Now, in order to avoid building the retrieved total database we do not evaluate $exp_{O_m}(q)$ on the retrieved total database. Instead, we transform $exp_{O_m}(q)$ to a new query $valid_E(exp_{O_m}(q))$ over the global relations on the basis of E and we use that query to access the underlying data integration systems. This is performed by the "*Valid Rewriter*" component which is also shown on Fig. 3. Bellow we describe the implementation of the aforementioned steps.

5.1 Query expansion.

In this step, the query is expanded to take into account the constraints coming from the ontology. Query expansion amounts to rewriting the query q posed to the ontology version O_m into a new query q', so that all the knowledge about the constraints in ontology has been "compiled" into q'. Recall that we consider an ontology as a schema with constraints. This is performed by constructing the *perfect rewriting* of q.

Definition 5.3 (Perfect Rewriting): *Let I an EDI system and let q be a query over O_m. Then q_p is called a perfect rewriting of q w.r.t. I if, for every global database $\mathfrak{D}$, $q^{I,\mathfrak{D}} = q_p^{ret(I,\mathfrak{D})}$.*

Algorithms for computing the perfect rewriting of a query q w.r.t to a schema, have been presented in [1, 9]. In our work, we use the QuOnto system [1] in order to produce the perfect rewriting of our initial query. Perfect rewriting is in our case PTIME in the size of ontology and NP in the size of query. For more genera classes of logic it is complete for PSPACE and 2EXPTIME as proved in [9]. Continuing our example if we expand q_1 we get q_2:

$\mathbf{q_2}$: $\pi_{?SSN,?NAME,?ADDRESS}$ (
(?X, type, Person) AND
(?X, ssn, ?SSN) AND
(?X, fullname, ?NAME) AND
(?X, has_contact_point, ?Y) AND
(?Y, type, Cont.Point) AND
(?Y, address, ?ADDRESS))

U

$\pi_{?SSN,?NAME,?ADDRESS}$ (
(?X, type, Actor) AND
(?X, ssn, ?SSN) AND
(?X, fullname, ?NAME) AND
(?X, has_contact_point, ?Y) AND
(?Y, type, Cont.Point) AND
(?Y, address, ?ADDRESS))

This is produced by considering the transitive constraint of the *subClass* relation among the classes "*Person*" and *"Actor"*.

5.2 Computing Valid Rewritings

Now instead of evaluating $exp_{O_m}(q)$ on the retrieved total database, we transform it to a new query called *valid rewriting*, i.e. $valid_E(exp_{O_m}(q))$. This is done as already discussed in order to avoid the construction the retrieved total database.

Definition 5.4 (Valid Rewriting): *Let I an EDI system and let q be a query over ret(I, $\mathfrak{D}$) . Then q_{valid} is called a valid rewriting of q w.r.t. ret(I, $\mathfrak{D}$) if, for every global database $\mathfrak{D}$, $q^{ret(I,\mathfrak{D})} = q_{valid}{}^{\mathfrak{D}}$.*

When the retrieved total database is produced by GAV mappings as in our case, query rewriting is simply performed using *unfolding* [1]. This is a standard step in data integration [2] which trivially terminates and it is proved that it preserves soundness and completeness [22]. Moreover, due to the disjointness of the input and the output alphabet, each GAV mapping acts in isolation on its input to produce its output. So we only need to scan the GAV mappings once in order to unfold the query and the time complexity of this step *O(N*M)* where *N* is the number of change operations in the evolution log and *M* is the number of sub-goals in the query.

Now, we can state the main result of this section.

Theorem 1 (Soundness and Completeness): *Let I and EDI system, q a query posed to I, $\mathfrak{D}$ a global database for I such that I is consistent w.r.t. $\mathfrak{D}$, and t a tuple of constants of the same arity as q. Then $t \in q^{I,\mathfrak{D}}$ if and only if $t \in [valid_E(exp(q))]^{\mathfrak{D}}$.*

Continuing our example, unfolding the query q_2 using the mappings mu_1, mu_3 and mu_3 will result to the following query: $\boldsymbol{q_3}$: $\pi_{?SSN,?NAME,?ADDRESS}$ *((?X, type, Actor) AND (?X, ssn, ?SSN) AND (?X, name, ?NAME) AND (?X, has_contact_point, ?Y) AND (?Y, type, Cont.Point) AND (?Y, street, ?ADDRESS) AND (?Y, city, ?ADDRESS))*

Finally, our initial query will be rewritten to the union of q_3 (issued to the data integration system that uses O_1) and q_2 (issued to the data integration system that uses O_2).

6 Discussion

So far we have described the scenario where we construct the change logs E^{O_m, O_i} between O_m and all O_i ($1 \leq i < m$) using the algorithm from [8]. Then we formulate a query q using the ontology version O_m, and we use the corresponding GAV mappings to produce and evaluate the $valid_E(exp_{O_m}(q))$

However, based on the composition property (proposition 1), we could avoid the computation of all those change logs from scratch each time. Instead, of constructing E^{O_m, O_i} for all i ($1 \leq i < m$) we could only construct all $E^{O_j, O_{j-1}}$ $(2 \leq j \leq m)$ between the subsequent ontology versions, thus minimizing the total construction cost[2] -since the compared ontologies now have more common elements. However, we have to keep in mind that the time of constructing a sequence of changes is spent only once during system setup.

Corollary 2: $E^{O_m, O_i} = \bigcup_i^{m-1} E^{O_{j+1}, O_j}$

Moreover, whenever a new ontology version occurs, we can construct the change log between the new ontology version and the previous ontology version - and not all change logs from scratch. Of course, this will lead to larger sequences of change logs, but will allow the uninterrupted introduction of new ontology versions to the system. Ideally, we would also like to accept queries formulated using ontology version O_1

[2] The complexity of the algorithm for input O_1, O_2 is $O(max(N_1, N_2, N^2))$.

and to rewrite it to the newer ontology versions. This would be really useful since in many systems queries might be stored and we wouldn't like to change them every time the ontology evolves. However, in order to achieve this we would have to use the inverse GAV mappings for query rewritings which are not always possible to produce. Our approach deals with the inversibility on the level of change operations and not at the logical level of the produced GAV mappings. So, instead of trying to produce the inverse of the initial GAV mappings, we invert the sequence of changes (which is always possible according to corollary 1) and then use the inverted sequence of changes to produce the GAV mappings that will be used for query rewriting to the current ontology version. This enhances the impact of our approach.

Actually, it now becomes obvious that it is straight forward to accept a query formulated in any ontology version O_i ($1 \leq i \leq m$) and to get the rewritings for all ontology versions using the inverted list of changes for the O_j that $j>i$.

Now, despite the fact that both query expansion and unfolding always terminate in our setting, problems may occur. Consider as an example the query q_4 that asks the "*gender*" and the "*name*" of an "*Actor*" using ontology version O_1: $\mathbf{q_4}$: $\pi_{?NAME,?GENDER}$ *((?X, type, Actor) AND (?X, name, ?NAME)* ***)*** *AND (?X, gender, ?GENDER*)***)***

Trying to rewrite the query q_4 to the ontology version O_2 our system will first expand it. Then it will consider the GAV mappings produced from the inverted sequence of changes (as they have been presented at the end of the sub-section 3b). So, the following query will be produced by unfolding using the mapping $\forall name, type(name, property) \rightarrow \exists fullname, type(fullname, property)$ - produced from *inv(*u_1*)*: $\pi_{?NAME,?GENDER}$*((?X, type, Actor) AND (?X,* ***fullname****, ?NAME*) *AND (?X, genter, ?GENDER*)***)***

However, it is obvious that the query produced will not provide any answers when issued to the data integration system that uses O_2 since the "g*ender"* literal no longer exist in O_2. This happens because the *inv(*u_4*)* change operation is not *information preserving change* among the ontology versions. It deletes information from the ontology version O_1 without providing the knowledge that this information is transferred on another part of the ontology. This is also the reason that low-level change operations (simple triple addition or deletion) are not enough to dictate query rewriting. Although, this might be considered as a problem, actually it is not, since if we miss the literal "*gender*" in version O_2 this would mean that we have no data in the underlying local databases for that literal. But even then, besides providing answers to the users for the data integration systems that can answer the corresponding rewritings, we provide to the users two more options:

The first option is to notify the user that some underlying data integration systems were not possible to answer their queries and present the reasons for that. For our example, our system will report that the data integration system that uses O_2 was not able to answer the initial query since the literal "*gender*" does not exist in that ontology version. To identify the change operations that lead to such a result we define the notion of *affecting change operations*.

Definition 6.1 (Affecting change operation)**.** *A change operation u affects the query q (with graph pattern G), denoted by u* $\Diamond$ *q, iff: 1)* $\delta_d(u) \neq \emptyset$*, 2)* $\delta_a(u) = \emptyset$ *and 3)* $\exists$ *triple pattern t in G that can be unified with a triple of* $\delta_d(u)$.

The first condition ensures that the operation deletes information from the ontology without replacing it with other information (condition 2), thus the specific change operation is not information preserving. However, we are not interested in general for the change operations that are not information preserving. We specifically target those change operations that change the ontology part which corresponds to our query (condition 3). In order to identify those change operations that affect user query, we have to scan the evolution log once for each query sub-goal. So the complexity of the corresponding algorithm is *O(N*M)* where *N* is the number of change operations in the evolution log and *M* is the number of sub-goals in the query. Users then can use that information in order to respecify only a specific set of their mappings if desired.

The second option is to produce *more general answers* for the data integration sub-systems that cannot answer input queries. Our solution here is that when a change operation *affects* a query rewriting, we can check if there is another triple *t'* (in the previous ontology version) which is the "*parent*" of the deleted triple *t*. If such a triple exists in the current ontology version we can ask for that triple instead, thus providing *a more general answer*.

Assume for example, an alternative ontology version O_1, where the "*personal_info*" property is a super-property of the "*gender*" property. Assume also the same sequence of changes from O_1 to O_2 (the list of inverted changes presented in Section 3.2). Then, if query q_4 previously described is issued, we are able to identify that the triple "*Actor, gender, xsd:String*" has been deleted and to look for a more general query. The query that our system produces, and that provides more general answer to user query is: q_5: $\pi_{?NAME,?GENDER}$ *((?X, type, Actor) AND (?X, fullname, ?NAME) AND (?X, personal_info, ?GENDER)*

Producing more general answers requires a slight extension of the algorithm for identifying affecting operations and is not presented due to space limitations.

7 Conclusions

In this paper, we argue that ontology evolution is a reality and data integration systems should be aware and ready to deal with that. To that direction, we presented a novel approach that allows query answering under evolving ontologies without mapping redefinition.

Our architecture is based on a module that can be placed on top of any traditional ontology-based data integration system, enabling ontology evolution. It does so by using high-level changes to model ontology evolution and uses those changes in order to rewrite not the mappings but the query itself among ontology versions. The process of query rewriting proceeds in two steps, namely *query expansion* and *valid rewriting*, and is proved to be effective, scalable and efficient. Experiments were performed, but not reported here due to space limitations; confirm the potential impact of our approach. To the best of our knowledge, no other system today is capable of automatically answering queries over multiple ontology versions.

As future work, several challenging issues need to be further investigated. For example, local schemata may evolve as well, and the structured DBMS data might be replaced with semi-structured ones. An interesting topic would be to extend our approach for OWL ontologies and to expand our approach to handle the full expressiveness of the SPARQL language. It becomes obvious that ontology evolution in data integration is an important topic and several challenging issues remain to be investigated in near future.

References

[1] Poggi, A., Lembo, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Linking data to ontologies. Journal on data semantics X (2008) 133-173

[2] Lenzerini, M.: Data integration: a theoretical perspective. SIGMOD (2002)

[3] Flouris, G., Manakanatas, D., Kondylakis, H., Plexousakis, D., Antoniou, G.: Ontology change: Classification and survey. Knowl. Eng. Rev. 23 (2008) 117-152

[4] Velegrakis, Y., Miller, J., Popa, L.: Preserving mapping consistency under schema changes. The VLDB Journal 13 (2004) 274-293

[5] Yu, C., Popa, L.: Semantic adaptation of schema mappings when schemas evolve. VLDB (2005)

[6] Curino, C.A., Moon, H.J., Ham, M., Zaniolo, C.: The PRISM Workwench: Database Schema Evolution without Tears. ICDE (2009) 1523-1526

[7] Kondylakis, H., Flouris, G., Plexousakis, D.: Ontology & Schema Evolution in Data Integration: Review and Assessment. ODBASE, OTM Conferences, (2009) 932-947

[8] Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V.: On Detecting High-Level Changes in RDF/S KBs. ISWC (2009) 473 - 488

[9] Cali, A., Gottlob, G., Lukasiewicz, T.: Datalog+-: a unified approach to ontologies and integrity constraints. ICDT. ACM, St. Petersburg, Russia (2009) 14-30

[10] Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.-C.: Composing schema mappings: Second-order dependencies to the rescue. ACM Trans. Database Syst. 30 (2005) 994-1055

[11] Barbosa, D., Freire, J., Mendelzon, A.O.: Designing information-preserving mapping schemes for XML. VLDB. VLDB Endowment, Trondheim, Norway (2005) 109-120

[12] Moro, M.M., Malaika, S., Lim, L.: Preserving XML queries during schema evolution. WWW. ACM, Banff, Alberta, Canada (2007) 1341-1342

[13] Rizzi, S., Golfarelli, M.: X-Time: Schema Versioning and Cross-Version Querying in Data Warehouses. ICDE (2007) 1471-1472

[14] Xuan, D.N., Bellatreche, L., Pierra, G.: A Versioning Management Model for Ontology-Based Data Warehouses. DaWaK, Vol. 4081. Springer, Krakow, Poland (2006) 195-206

[15] Bounif, H.: Schema Repository for Database Schema Evolution. DEXA, (2006) 647-651

[16] Edelweiss, N., Moreira, A.F.: Temporal and versioning model for schema evolution in object-oriented databases. Data Knowl. Eng. 53 (2005) 99-128

[17] Moon, H.J., Curino, C.A., Zaniolo, C.: Scalable architecture and query optimization for transaction-time DBs with evolving schemas. SIGMOD 2010 ACM, USA (2010) 207-218

[18] Theoharis, Y.: On Graph Features of Semantic Web Schemas. IEEE Transactions on Knowledge and Data Engineering 20 (2007) 692-702

[19] Brickley, D., Guha, R.: {RDF Vocabulary Description Language 1.0: RDF Schema}. WWW (2004)

[20] Cali, A., Martinenghi, D.: Querying the deep web. EDBT. (2010) 724-727

[21] Perez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34 (2009) 1-45

[22] Calì, A., Calvanese, D., Giacomo, G.D., Lenzerini, M.: Data Integration under Integrity Constraints. Advanced Information Systems Engineering (2006) 262-279