Handling Redundant Processing in OBDA Query Execution Over Relational Sources

This is a pre-print of an article published in Journal of Web Semantics.
The final authenticated version is available online at:
https://doi.org/10.1016/j.websem.2021.100639

Handling Redundant Processing in OBDA Query Execution Over Relational Sources

Dimitris Bilidas[a,*], Manolis Koubarakis[a]

[a]Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Panepistimiopolis, Ilissia, Athens 15784, Greece

Abstract

Redundant processing is a key problem in the translation of initial queries posed over an ontology into SQL queries, through mappings, as it is performed by ontology-based data access systems. Examples of such processing are duplicate answers obtained during query evaluation, which must finally be discarded, or common expressions evaluated multiple times from different parts of the same complex query. Many optimizations that aim to minimize this problem have been proposed and implemented, mostly based on semantic query optimization techniques, by exploiting ontological axioms and constraints defined in the database schema. However, data operations that introduce redundant processing are still generated in many practical settings, and this is a factor that impacts query execution. In this work we propose a cost-based method for query translation, which starts from an initial result and uses information about redundant processing in order to come up with an equivalent, more efficient translation. The method operates in a number of steps, by relying on certain heuristics indicating that we obtain a more efficient query in each step. Through experimental evaluation using the Ontop system for ontology-based data access, we exhibit the benefits of our method.

Keywords: Query Translation, Data Integration, Ontology-Based Data Access, Ontop

1. Introduction and Outline

Ontology Based Data Access (OBDA) is a database technique in which an ontology is linked to underlying data sources through mappings. An end user can pose queries over the ontology, which we assume to represent a familiar vocabulary and conceptualization of the user domain. The OBDA system automatically translates the query and sends it for execution to the underlying data sources. This approach provides the end user with a convenient abstraction over possibly complex schemas and details about the data storage and query processing. The query translation involves query rewriting and query unfolding. During query rewriting, an initial query over an ontology is rewritten in order to take into consideration the ontological axioms. The result of this process is a query that when posed over property and class assertions only (that is, by disregarding all the ontological axioms), will return the same answers as the initial query posed over the ontology. This is done using the notion of certain answers, that is, answers present in every model of the ontology. During query unfolding the rewritten query is transformed into another query expressed in the query language of the underlying data sources. In what follows we consider an OBDA setting, where an OWL 2 QL ontology is linked through mappings to data stored in a relational database management system (RDBMS). This method provides the user with access to a virtual RDF graph. The original query is a conjunctive query expressed over the vocabulary of the virtual RDF graph, and the result of rewriting and unfolding is a SQL query.

Example 1. As an example of OBDA setting consider a relational schema that contains the relational tables A_1, A_2, A_3, C_1 and C_2 and the mappings from Figure 1. In these mappings P_1, Q_1, R_1, P_2, P_3

*Corresponding author

Email addresses: d.bilidas@di.uoa.gr (Dimitris Bilidas), koubarak@di.uoa.gr (Manolis Koubarakis)

and Q_3 are properties defined in the ontology, whereas f, g, h and k are functions that construct ontology objects from database values. These functions are responsible for constructing an object that acts as an ontology individual out of values occurring in the database. In our setting, they construct an RDF term. A query posed over the ontology can be the following: $ans(x, y, w, z) \leftarrow P_1(x, y), P_2(x, w), P_3(y, z)$.

The notion of OBDA as we describe it, was presented in [21]. There, the result of query rewriting of an initial conjunctive query (CQ) over the ontology is a union of conjunctive queries (UCQ) over the vocabulary of the ontology. Then, the authors define a faithful representation of this UCQ, along with the mappings and database instance in terms of a logic program. Query unfolding is based on partial evaluation of such logic programs, and as final result it produces a query which can be viewed as an SQL query. More details about this process are given in Section 3. Subsequent research was focused on more efficient rewritings in the form of UCQs over the ontology [14, 6, 20]. The main aim of these approaches was to produce a UCQ with as few subqueries as possible, as it was observed that the number of union subqueries in the result of query rewriting could be very large. A different approach was followed in [3], where a cost-based comparison of different reformulations is carried out, considering that no mappings are used and the ABox is directly stored in the external database. In general, the final query in this case will be an SQL query that contains joins over UCQs (JUCQs). An extension of this work for arbitrary relational schemas, so that it also takes into consideration the unfolding step with arbitrary mappings, is presented in [16].

Regarding the implementation of OBDA systems, it has been observed that in practice it is more efficient to compile ontological knowledge regarding class and property hierarchies into the mappings, and ignore such axioms during query rewriting. For this reason, Ultrawrap-OBDA[26] uses the notion of saturated mappings and Ontop[4] uses the so called $\mathcal{T}$-Mappings [22]. For example, consider the setting of Example 1 and an ontology that contains the following axioms: $Q_1 \sqsubseteq P_1, R_1 \sqsubseteq P_1$ and $Q_3 \sqsubseteq P_3$. We can ignore the axiom $Q_1 \sqsubseteq P_1$ during rewriting if we add to the original mappings the mapping $m1'$ from Figure 2, and similar for the other two axioms.

In [22] three main reasons are specified for the presence of a large number of union subqueries in the result of query translation: i) ontological queries with existentially quantified variables that can lead to rewritings of exponential size, ii) large ontological hierarchies and iii) multiple mappings for each ontology term. Also, the authors notice that the first reason is rarely observed in real-world ontologies and queries. As a result, when compiling ontological information about hierarchies into the mappings, as for example in the Ontop $\mathcal{T}$-mappings, the last two important reasons that lead to a large number of subqueries are encountered during query unfolding. As an example, consider the query from Example 1 posed over the previously specified OBDA setting and $\mathcal{T}$-mappings. The unfolding method from [21] will produce a UCQ over the database that contains six union subqueries as shown in Figure 3. Each subquery corresponds to a different combination of the three mappings defined for P_1 with the two mappings defined for P_3. One can easily see that in case of queries with many atoms posed over large hierarchies, the final UCQ can contain hundreds or thousands of subqueries. On the other hand, a different unfolding method could choose to first compute as intermediate results the queries that correspond exactly to the first and third atoms of the initial query. In the specific example, the first temporary result would be a union query over tables A_1, A_2 and A_3 and the second temporary result would be a union query over tables C_1 and C_2. The final result would be a join of UCQs. Finally, one could choose an intermediate strategy, that would compute only one of these two intermediate results. Clearly, a cost-based decision should be made by the OBDA system regarding which exactly of these intermediate results should be computed, and if the overhead from computing and saving these results is counterbalanced from the gain in the final query.

Unfortunately, uncertainty about query execution costs is an inherent problem in data integration, where the mediator system (in our case the OBDA system) operates outside the database engine[11], as knowing all the factors that affect query execution is difficult or even impossible. For example, these factors include the exact execution plan that will be chosen by the RDBMS, including the access methods for each base relation and the join order in a join query, hardware characteristics like the amount of available memory and disk throughput, the disk block size, the exact details of the database physical design, like the existing indexes and the kind of each index and several other

$$
\begin{aligned}
m1 &: A_1(v_1^{m1}, v_2^{m1}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1})) \\
m2 &: A_2(v_1^{m2}, v_2^{m2}) \rightarrow Q_1(f(v_1^{m2}), g(v_2^{m2})) \\
m3 &: A_3(v_1^{m3}, v_2^{m3}, v_3^{m3}) \rightarrow R_1(f(v_1^{m3}), g(v_2^{m3})) \\
m4 &: A_3(v_1^{m4}, v_2^{m4}, v_3^{m4}) \rightarrow P_2(f(v_1^{m4}), h(v_3^{m4})) \\
m5 &: C_1(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5})) \\
m6 &: C_2(v_1^{m6}, v_2^{m6}) \rightarrow Q_3(g(v_1^{m6}), k(v_2^{m6}))
\end{aligned}
$$

Figure 1: Example Mappings

$$
\begin{aligned}
m1 &: A_1(v_1^{m1}, v_2^{m1}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1})) \\
m1' &: A_2(v_1^{m1'}, v_2^{m1'}) \rightarrow P_1(f(v_1^{m1'}), g(v_2^{m1'})) \\
m1'' &: A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}) \rightarrow P_1(f(v_1^{m1''}), g(v_2^{m1''})) \\
m2 &: A_2(v_1^{m2}, v_2^{m2}) \rightarrow Q_1(f(v_1^{m2}), g(v_2^{m2})) \\
m3 &: A_3(v_1^{m3}, v_2^{m3}, v_3^{m3}) \rightarrow R_1(f(v_1^{m3}), g(v_2^{m3})) \\
m4 &: A_3(v_1^{m4}, v_2^{m4}, v_3^{m4}) \rightarrow P_2(f(v_1^{m4}), h(v_3^{m4})) \\
m5 &: C_1(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5})) \\
m5' &: C_2(v_1^{m5'}, v_2^{m5'}) \rightarrow P_3(g(v_1^{m5'}), k(v_2^{m5'})) \\
m6 &: C_2(v_1^{m6}, v_2^{m6}) \rightarrow Q_3(g(v_1^{m6}), k(v_2^{m6}))
\end{aligned}
$$

Figure 2: Example $\mathcal{T}$-Mappings

factors.

On the other hand, one could expect that the RDBMS is capable of optimizing the produced query, since it performs query planning and optimization by taking into consideration the aforementioned parameters. Unfortunately, database engines focus on optimization of certain aspects of queries, including join ordering of multi-join queries, optimization of aggregate functions, access methods for each relation, etc. Queries produced by OBDA systems have some characteristics that are not regularly encountered on human-written queries for database applications. One such characteristic is the occurrence of common subexpressions in different parts of the query, for example in different subqueries of a union query. As we saw, the number of these subexpressions and subqueries can be very large. Although common subexpression identification (and in the case of multiple queries the related multi-query optimization area) have long been investigated in database research and implemented in database prototypes [25, 19, 24], to the best of our knowledge these methods have not become integral part of commercial RDBMSs, due to the increase in optimization time and the complexity introduced to the query optimizer. But since these common subexpressions are created during query translation, the OBDA system has the knowledge about them that can be taken into consideration to produce the final SQL query. Furthermore, it has been observed [16] that by using knowledge from the mappings, we can compute during system setup some parameters that will help us obtain more accurate selectivity estimations. For example, in our approach, a crucial factor that must be used when deciding about the exact form of the final SQL query, is the number of duplicates contained in the mappings used during unfolding for each ontology predicate. For example, for predicate P_1 of the query given in the previous example, it is crucial to know the number of duplicates rows in tables A_1, A_2 and the table obtained by selecting the first two columns of table A_3. The OBDA system knows from the mappings for which such

$$
\begin{aligned}
ans(f(x), g(y), h(w), k(z)) \leftarrow \\
A_1(x, y), A_3(x, v_1, w), C_1(y, z) \vee \\
A_2(x, y), A_3(x, v_1, w), C_1(y, z) \vee \\
A_3(x, y, v_2), A_3(x, v_1, w), C_1(y, z) \vee \\
A_1(x, y), A_3(x, v_1, w), C_2(y, z) \vee \\
A_2(x, y), A_3(x, v_1, w), C_2(y, z) \vee \\
A_3(x, y, v_2), A_3(x, v_1, w), C_2(y, z)
\end{aligned}
$$

Figure 3: UCQ over the database

columns and tables it should collect such information as an one-time task prior to query execution. On the other hand, an RDBMS cannot accurately estimate the number of duplicates in seemingly unrelated tables and columns during query execution.

Given the previous observations, in this work we propose a cost-based method employed by the OBDA system during unfolding for choosing the final form of the SQL query to be executed by the RDBMS. This method relies on heuristics that in turn rely only on factors known to the OBDA system, such as sizes of the relations, duplicates introduced by the mappings for each ontology term and selectivity estimation for simple CQs over the database, that are not affected by issues such as join ordering or access methods, and thus can be performed even from a system operating outside the RDBMS as long as some basic statistics about the tables have been obtained prior to the deployment of the system. Specifically, our method starts with the "fully" unfolded query produced by the method of [21] as the baseline, and uses the heuristics in order to "fold" back specific paths, when this is expected to be more effective. Each such fold corresponds to the creation of an intermediate table, as explained in the previous example. These heuristics are based on the notion of redundant processing between the union subqueries. We make a distinction between two kinds of redundant processing: i) duplicate answers and ii) repeated operations (disk access regarding the same data) from different union subqueries of the same query even in the absence of duplicate answers.

Regarding duplicates, using the standard set semantics for queries over ontologies, the final answer should be duplicate-free, but since RDBMSs operate using the bag semantics, duplicates are often introduced during query evaluation. Duplicates can be introduced as different ways to obtain the same fact from the data, for example the same tuple may be produced from different mappings used for the same property or class assertion. Using the unfolding method from [21], this will result in duplicate answers coming from different union subqueries. But duplicates can be introduced even from a single mapping due to the projection operator in the SQL query in the body of the mapping. In this setting, duplicates are redundant answers whose impact can be detrimental for query evaluation, as the size of intermediate results can increase exponentially in the number of joins in the query. Even if the final SQL query produced by an OBDA system dictates that the result should be duplicate-free using the SQL DISTINCT or UNION keyword, relational systems rarely consider early duplicate elimination in order to limit the size of intermediate results, but only perform the task on the final query result. This behavior is justified by the fact that duplicate elimination is a costly blocking operation [2] and also that the SQL queries are usually formulated by expert users who take into consideration the integrity constraints of normalized relational schemas. Under these assumptions, considering early duplicate elimination options during optimization is not usually regarded worthy. Contrary to this situation for SQL queries, it has been ascertained [13] that in real-world OBDA settings, duplicate answers frequently dominate query results and also that this appears as "noise" to end users that might be using a visual query formulation tool. In the previous version of this work [1], we introduced a heuristic regarding early duplicate elimination, for duplicates introduced from a single mapping. In this version we extend this heuristic for the case of duplicates that show up in different union subqueries, and use it to help us decide when to "fold" back specific

branches of the unfolded query.

Regarding the second kind of redundant processing, this depends heavily on the exact execution plan that will be chosen by the RDBMS. As an example, consider the UCQ from Figure 3 and let us suppose that there are no duplicates (each fact for each ontology predicate can be obtained only once from a single mapping). Also suppose that the RDBMS chooses to perform all the joins using index-based nested loops, using for the first three subqueries the table C_1 as the leftmost table and for the next three subqueries the table C_2 as the leftmost table. In this case, the redundant processing is equal to the two scans of table C_1 plus the two scans of table C_2 (ignoring the possible impact of the memory cache). If there was no redundant processing, then it is reasonable to assume that this form of the query would be the most efficient translation, as it consists of simple CQs which the RDBMS can efficiently optimize and probably evaluate in parallel. But since we have redundant processing, one would expect that it would be more efficient to first compute and save the temporary union table corresponding to the three mappings for P_1, if the RDBMS will again choose to perform index-based nested loops and the cost for creating and saving the temporary result is smaller than the cost of the initial redundant processing. As all these possible execution plans cannot be known to the OBDA system, for this case of redundant processing, we use a criterion according to which temporary tables are created in a "conservative" manner, only when it is almost certain that this decision will lead to smaller execution cost.

In this work we present efficient solutions to the problem of handling redundancy, considering ontologies belonging to the OWL 2 QL language[1], as the W3C recommendation for query answering against datasets stored in relational back-ends. Nevertheless, several aspects of this work can be considered for other ontology languages as well. As mentioned, an early version of this work was presented in [1], where a heuristic was presented for early duplicate elimination in duplicates introduced from a single mapping (that is for each union subquery of the final SQL query separately). This heuristic was evaluated over four different RDBMSs and it was shown that its usage is justified and that for query mixes from two different used benchmarks, such that low selectivity queries do not dominate execution time, it can lead to overall improvement of up to 25% compared to the strategy of always performing duplicate elimination. The main contributions of the present work, extending this previous version in several aspects, are as follows:

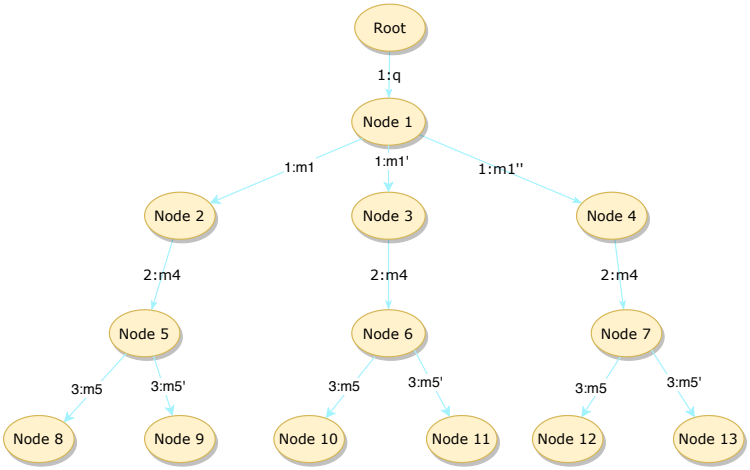
- We enhance the unfolding step previously described in the literature with cost-based decisions regarding the redundant processing, obtaining a full cost-based method for OBDA query translation (Section 3).
- We extend the heuristic in order to deal with duplicate answers coming from different union subqueries (Section 4).
- We take into consideration other forms of redundant processing in the form of repeated operations (Section 4).
- We implement our method for cost-based translation by modifying the state of the art OBDA system Ontop [22] and we perform extended experimental evaluation (Section 5).

The organization of this paper is as follows. We start by providing some preliminaries regarding ontologies, mappings, relational databases and logic programs (Section 2). In Section 3 we modify the unfolding method from [21], which is based on partial evaluation of logic programs, in order to explore equivalent results given that certain mappings have been replaced by a combined mapping which we define. In Section 4 we describe the cost-based decisions and we present the algorithm that incorporates them in the unfolding process. In Section 5 we present experimental evaluation of our implementation using the Ontop OBDA framework and the NPD and LUBM benchmarks. We also use the Wisconsin benchmark to compare our results with the results of [16]. In Section 6 we present relevant work and conclusions.

2. Preliminaries

We consider the following pairwise disjoint alphabets: Σ_O of ontology predicates, Σ_R of database relation predicates, Const of constants, Var of variables and Λ of function symbols, where each function symbol has an associated arity. We also consider that Const is partitioned into $\mathit{DB}_{\mathit{Const}}$ of database constants and $\mathit{O}_{\mathit{Const}}$ of ontology constants.

[1] https://www.w3.org/TR/owl2-profiles/



$$Root : ans(x,y,z)\theta_0 \leftarrow ans(x,y,z)$$
$$\theta_0 = \{\}$$
$$Node1 : ans(x,y,z)\theta_0\theta_1 \leftarrow P_1(x,y), P_2(x,h(A)), P_3(y,z)$$
$$\theta_1 = \{\}$$
$$Node2 : ans(x,y,z)\theta_0\theta_1\theta_2 \leftarrow A_1(v_1^{m1}, v_2^{m1}), P_2(f(v_1^{m1}), h(A)), P_3(g(v_2^{m1}), z)$$
$$\theta_2 = \{x/f(v_1^{m1}), y/g(v_2^{m1})\}$$
$$Node3 : ans(x,y,z)\theta_0\theta_1\theta_3 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), P_2(f(v_1^{m1'}), h(A)), P_3(g(v_2^{m1'}), z)$$
$$\theta_3 = \{x/f(v_1^{m1'}), y/g(v_2^{m1'})\}$$
$$Node4 : ans(x,y,z)\theta_0\theta_1\theta_4 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), P_2(f(v_1^{m1''}), h(A)), P_3(g(v_2^{m1''}), z)$$
$$\theta_4 = \{x/f(v_1^{m1''}), y/g(v_2^{m1''})\}$$
$$Node5 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), P_3(g(v_2^{m1}), z)$$
$$\theta_5 = \{v_1^{m_4}/v_1^{m1}, v_3^{m_4}/A\}$$
$$Node6 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), P_3(g(v_2^{m1'}), z)$$
$$\theta_6 = \{v_1^{m_4}/v_1^{m1'}, v_3^{m_4}/A\}$$
$$Node7 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), P_3(g(v_2^{m1''}), z)$$
$$\theta_7 = \{v_1^{m_4}/v_1^{m1''}, v_3^{m_4}/A\}$$
$$Node8 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_8 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), C_1(v_2^{m1}, v_2^{m5})$$
$$\theta_8 = \{v_1^{m5}/v_2^{m1}, z/k(v_2^{m5})\}$$
$$Node9 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_9 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), C_2(v_2^{m1}, v_2^{m5'})$$
$$\theta_9 = \{v_1^{m5'}/v_2^{m1}, z/k(v_2^{m5'})\}$$
$$Node10 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{10} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), C_1(v_2^{m1'}, v_2^{m5})$$
$$\theta_{10} = \{v_1^{m5}/v_2^{m1'}, z/k(v_2^{m5})\}$$
$$Node11 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{11} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), C_2(v_2^{m1'}, v_2^{m5'})$$
$$\theta_{11} = \{v_1^{m5'}/v_2^{m1'}, z/k(v_2^{m5'})\}$$
$$Node12 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{12} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), C_1(v_2^{m1''}, v_2^{m5})$$
$$\theta_{12} = \{v_1^{m5}/v_2^{m1''}, z/k(v_2^{m5})\}$$
$$Node13 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{13} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), C_2(v_2^{m1''}, v_2^{m5'})$$
$$\theta_{13} = \{v_1^{m5'}/v_2^{m1''}, z/k(v_2^{m5'})\}$$

Figure 4: SLD Tree

As in [21], we use functions with symbols from Λ in order to solve the so called impedance mismatch problem of constructing ontology objects from values occurring in the database. We assume that for $\lambda_1, \lambda_2 \in \Lambda$, where $\lambda_1 \neq \lambda_2$, the range of function with symbol λ_1 and the range of function with symbol λ_2 are disjoint. That is, the same ontology object cannot be produced from different functions.

2.1. Databases.

We start by giving definitions for database instances and queries over them, following the bag semantics from [5]. A bag B is a pair (US_B, μ), where US_B is a set called the underlying set of B and μ is a function from elements of US_B to the positive integers, which gives the multiplicities of elements of US_B in B. A relation instance is a bag of tuples of fixed arity using constants from DB_{Const}. A source schema S is a set of relation names from Σ_R. A database instance D for a source schema S is a mapping from relation names in S to relation instances.

2.2. Queries.

We define queries following the bag semantics of [5]. In our definitions we use the term "SQL query" although the syntax of our formulas is that of first-order logic. Similarly, relation instances are viewed as bags of ground atoms (i.e., with no variables) of first-order logic.

A SQL query over a relational schema S is an expression that has the form: $SQL(\vec{x}) \leftarrow \alpha$, where α is a first order expression containing predicates from Σ_R, which are among the relations that belong to S, $SQL \in \Sigma_R$, $SQL \notin S$ and $\vec{x}$ is a vector of constants from DB_{Const} and variables from Var that appear in α.

A conjunctive query Q over a relational schema S is a SQL query, where α has the form $R_1(\vec{x_1}) \wedge ... \wedge R_n(\vec{x_n})$, where $\vec{x_1}, ..., \vec{x_n}$ are vectors of constants from DB_{Const} and variables from Var, and $R_1, ..., R_n \in S$. Variables from $\vec{x_1}, ..., \vec{x_n}$ that do not appear in $\vec{x}$ are existentially quantified, but we omit the quantifiers in order to simplify the reading. CQs roughly correspond to SQL Select-From-Where queries.

An assignment mapping of a conjunctive query Q into a database instance D is an assignment of values from DB_{Const} belonging to D to the variables of Q such that every atom in the body of Q is mapped to a ground atom in D. Let θ be an assignment mapping of Q into database instance D and let X be a variable in Q. We denote by $\theta(X)$ the constant in DB_{Const} to which θ maps X and we denote by $\theta(R_i(\vec{x_i}))$ the ground atom to which $R_i(\vec{x_i})$ is mapped.

Let μ_i denote the multiplicities $\mu(\theta(R_i(\vec{x_i}))), i = 1, ..., n$. The result due to θ of a conjunctive query Q over D is the tuple $(\theta(\vec{x}), \mu_\theta)$ with the multiplicity $\mu_\theta = \mu_1 \mu_2 \cdots \mu_n$. The result of a conjunctive query Q over a database instance D denoted by $Q(D)$ is given by $\uplus_\theta r_\theta$, where θ is any assignment mapping of Q into D, r_θ is the result due to θ and $\uplus$ denotes bag union.

2.3. Ontology and Mappings.

A TBox is a finite set of ontology axioms. An ABox is a finite set of membership assertions $A(\rho)$ or role assertions $P(\rho, \rho')$, where $\rho, \rho' \in O_{Const}$ and $A, P \in \Sigma_O$ denote a concept name and role (or property) name respectively. A DL ontology O is a pair $\langle \mathcal{T}, \mathcal{A} \rangle$ where $\mathcal{T}$ is a TBox and $\mathcal{A}$ an ABox.

A mapping assertion (or simply a mapping) m from a source schema S to a TBox $\mathcal{T}$ has the form: $\phi(\vec{x}) \rightarrow \psi$, where $\phi(\vec{x})$ will be denoted by body(m) and it is the right-hand side of an SQL query over a database schema S, ψ has the form $P(f^1(\vec{x^1}), f^2(\vec{x^2}))$ or $C(f^1(\vec{x^1}))$ with P (respectively C) $\in \Sigma_O$ a property (respectively concept) name, all variables in ψ also appear in $\vec{x}$, and each $f^j \in \Lambda$ is a function with arity equal to the length of $\vec{x^j}$ and range a subset of O_{Const}. The right-hand side will be denoted by head(m). A mapping collection $\mathcal{M}$ is a finite set of such mapping assertions. In this setting, having a conjunction of atoms in the head of the mapping assertion does not add to the expressivity of the mapping language [21].

Let $\mathcal{M}$ be a mapping collection, we will use the symbol $\mathcal{M}_{CQ}$ to denote the assertions from $\mathcal{M}$ whose body is a CQ over the database schema. In correspondence with CQs over a relational schema, we define a CQ over an ontology O as an expression of the form: $Query(\vec{x}) \leftarrow P_1(\vec{x_1}) \wedge ... \wedge P_n(\vec{x_n})$ where $\vec{x_1}, ..., \vec{x_n}$ are vectors of constants from O_{Const} and variables from Var, $\vec{x}$ is a vector of constants from O_{Const} and variables from Var that appear in $\vec{x_1}, ..., \vec{x_n}$, and $P_1, ..., P_n \in \Sigma_O$ are ontology predicates that appear in O. A union of conjunctive queries UCQ over an ontology O is an expression of the form $Query(\vec{x}) \leftarrow CQ_1(\vec{x}) \vee ... \vee CQ_n(\vec{x})$, where each CQ_i for $i = 1, ..., n$ is an expression of the form $P_1^i(\vec{x_1^i}) \wedge ... \wedge P_n^i(\vec{x_n^i})$ as in the previous definition.

2.4. Logic Programs

Following [21], we use partial evaluation of logic programs in order to translate a UCQ over the vocabulary of the ontology into a UCQ over the data sources. In this section we present basic notions from logic programs[17] regarding partial evaluation [18]. As we are interested in the translation of UCQs, we do not deal with negation, and as a result we only present notions related to definite logic programs. As a result, in what follows we are referring to definite programs, clauses and rules.

A logic program is a set of statements that have the following form: $\forall \vec{x}(A \leftarrow A_1 \wedge ... \wedge A_n)$, where $A, A_1, ..., A_n$ are atoms as in standard first order logic definitions and $\vec{x}$ are all the variables occurring in $A, A_1, ..., A_n$. Each such statement is also called a program clause, or a rule, with A being the head of the rule, and $A_1 \wedge ... \wedge A_n$ the body of the rule. A goal is a clause such that the head is empty. Following the standard convention in logic programming, we omit the existential quantifiers and use the syntactic form $A_1, ..., A_n$ for the body, instead of $A_1 \wedge ... \wedge A_n$, both in clauses and goals.

A substitution θ is a finite set of the form: $\{x_1/t_1, .., x_n/t_n\}$, where each x_i is a variable, each t_i is a term distinct from x_i, variables $x_1, ..., x_n$ are pairwise distinct and no variable x_i occurs in some term t_i. Let Exp be an expression. The application of a substitution θ on Exp is denoted $Exp\theta$ and is the expression obtained by Exp after replacing each occurrence of x_i with t_i for $i = 1, ..., n$. Let Exp_1 and Exp_2 be expressions. A unifier for Exp_1 and Exp_2 is a substitution θ such that $Exp_1\theta = Exp_2\theta$. Let $\theta_1 = \{x_1/s_1, ..., x_m/s_m\}$ and $\theta_2 = \{y_1/t_1, ..., y_n/t_n\}$ be substitutions such that no variable from $x_1, ..., x_m$ occurs in θ_2. The composition of θ_1 with θ_2 is the following substitution: $\{x_1/s_1\theta_2, ..., x_m/s_m\theta_2, y_1/t_1, ..., y_n/t_n\}$. The most general unifier (mgu) of two expressions Exp_1 and Exp_2, is a unifier ξ such that for every unifier ν of Exp_1 and Exp_2 there exists a substitution θ such that ν is the composition of ξ with θ.

A computation rule is a function from a set of goals to a set of atoms, such that the value of the function for a goal is always an atom, called the selected atom, in that goal.

Let G be $\leftarrow A_1, ..., A_m, ..., A_k$, C be $A \leftarrow B_1, ..., B_q$ and R be a computation rule. Then, the goal G' is derived from G and C using the mgu θ via R if the following conditions hold:

- A_m is the selected atom in G given by R,
- θ is an mgu of A_m and A,
- G' is the goal $\leftarrow (A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1}, ..., A_k)\theta$.

A resultant is a first order formula of the form $Q_1 \leftarrow Q_2$, where each of Q_1, Q_2 is either absent or a conjunction of atoms. Any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.

Let P be a program, G' be a goal with body G and R a computation rule. Then, the SLD-tree of $P \cup \{G'\}$ via R is the tree defined as follows:

- Each node is a resultant (possibly with an empty body)
- The root node is $G_0\{\} \leftarrow G_0$, where $G_0 = G$.
- Let $G\theta_0...\theta_i \leftarrow A_1, ..., A_m, ..., A_k$ be a node in the tree with $k \geq 1$ and suppose that A_m is the selected atom of the derivation given by R. Then, this node has a descendant for each input clause of $A \leftarrow B_1, ..., B_q$ of P such that A_m and A are unifiable. The descendant is $G\theta_0...\theta_{i+1} \leftarrow (A_1, ..., B_1, ..., B_q, ..., A_k)\theta_{i+1}$, where θ_{i+1} is an mgu of A and A_m.
- Nodes which are resultants with empty bodies have no descendants.

Each branch of the SLD-tree is a derivation of G'. A branch which ends in a node such that the selected atom does not unify with the head of any program clause is called a failure branch. A branch which ends in the empty clause is called a success branch. An SLD-tree is complete if all of its branches are either failure or success branches. An SLD-tree that is not complete is called partial.

In general an SLD-tree can contain branches that correspond to infinite derivations, but we will not deal with this case, as the logic programs that we will construct do not contain recursion.

The computed answer θ for a node $Q\theta_0, ..., \theta_i \leftarrow Q_i$ of an SLD-tree is the restriction of $Q\theta_0, ..., \theta_i$ to the variables in the goal G'.

Let P be a program, A an atom and R a computation rule and T an SLD-tree for $P \cup \{\leftarrow A\}$ via R. Then:

- any set of nodes such that each non-failing branch of T contains exactly one of them is a Partial Evaluation (PE) of A in P;

- the logic program obtained from P by replacing the set of clauses in P whose head contains A with a PE of A in P is a PE of P with respect to A.

The semantics of a logic program P can be defined by two different ways, proved to be equivalent. The first one is the declarative, that uses the model-theoretic semantics of first-order logic, where the semantics are given by the least Herbrand model, which contains the facts that are true in every model of P. The second way is the procedural, where the SLD-tree is used, and the semantics are given by the success set of P, that is all the facts A such that the SLD-tree of $P \cup \{\leftarrow A\}$ has a success branch. Also, it is known that the semantics of a program P coincide with the semantics of any partial evaluation of P[17].

3. Unfolding Queries Through Partial Evaluation

In this section we describe the process of unfolding queries over the ontology, into queries over the external relational database using mappings. We are following the approach of [21], with the following modifications:

- We enforce that during each step of the SLD-Derive process, the algorithm employs the computation rule that chooses for unification the leftmost possible atom in the right-hand side of the resultant.
- We make a distinction between mapping assertions whose body is a CQ over the database and the rest of the mapping assertions.
- We define a step that "folds" back specific branches of the PE tree based on the notion of combined mapping, and we show that the SQL query that is obtained based on this form of the PE tree has exactly the same answers with the SQL query obtained using the initial form of the tree.

The logic program for a UCQ $Q(\vec{x}) \leftarrow CQ_1(\vec{x}) \vee ... \vee CQ_n(\vec{x})$ over: (i) an ontology $\mathcal{O}$ (ii) a database instance D over a database schema S and (iii) a mapping collection $\mathcal{M}$ from source schema S to the vocabulary of $\mathcal{O}$ is defined in [21]. As it is shown that the result of the unfolding process is independent of the database instance D, here we omit the second component and we directly define the logic program with respect to $\mathcal{O}$ and $\mathcal{M}$. Also, we modify the process by using auxiliary predicates only for mapping assertions in $\mathcal{M} \setminus \mathcal{M}_{CQ}$.

The program for Q and $\mathcal{M}$, denoted $P(Q, \mathcal{M})$ is the logic program defined as follows:

- $P(Q, \mathcal{M})$ contains the clause $Q(\vec{x}) \leftarrow CQ_i(\vec{x})$ for each CQ_i in the right-hand side of Q.
- $P(Q, \mathcal{M})$ contains each mapping assertion $m \in \mathcal{M}_{CQ}$.
- For each mapping assertion $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$, $P(Q, \mathcal{M})$ contains the clause $head(m) \leftarrow Aux_m(\vec{x})$, where Aux_m is an auxiliary predicate associated to m, whose arity is the same as $head(m)$.

We now present the function SLD-Derive defined in [21], with the extra condition that we enforce use of the computation rule that chooses for unification the leftmost possible atom. The SLD-Derive($P(Q, \mathcal{M})$) takes as input $P(Q, \mathcal{M})$, where Q has the form $q(x) \leftarrow \beta$, and returns a set Res of resultants constituting a PE of $q(\vec{x})$ in $P(Q, \mathcal{M})$, by constructing an SLD-tree for $P(Q, \mathcal{M}) \cup \{\leftarrow q(\vec{x})\}$ as follows:

- it starts by selecting the atom $q(\vec{x})$,
- it continues by selecting the atoms whose predicates belong to the alphabet of $\mathcal{T}$, as long as possible, using the computation rule R which selects each time the leftmost such atom
- it stops the construction of a branch when no atom with predicate in the alphabet of $\mathcal{T}$ can be selected.

The partial evaluation $PE(Q, \mathcal{M})$ of $P(Q, \mathcal{M})$ with respect to $q(\vec{x})$ is obtained by dropping the clauses for q in $P(Q, \mathcal{M})$ and replacing them with the result of SLD-Derive($P(Q, \mathcal{M})$).

Example 2. Consider the query $ans(x, y, z) \leftarrow P_1(x, y), P_2(x, h(A)), P_3(y, z)$, with $h \in \Lambda$ and $A \in DB_{Const}$ the mapping collection ($\mathcal{T}$-mappings) shown in Figure 2 and a database instance over a schema that contains the relation names A_1, A_2, A_3, C_1 and C_2 with tuples of appropriate arities according to Figure 2. The SLD-tree for $P(Q, \mathcal{M}) \cup \{\leftarrow ans(x, y, z)\}$ is shown in Figure 4.

In [21] the virtual ABox given by a mapping collection $\mathcal{M}$ over a database instance D for a database

schema S is defined as the set of ABox assertions generated by applying each mapping assertion in $\mathcal{M}$ over D and it is shown that for each tuple of constants $\vec{t}$, $P(Q, \mathcal{M}) \cup \{\leftarrow q(\vec{t})\}$ is unsatisfiable if and only if $\vec{t}$ belongs to the result of executing Q over the database instance that stores exactly the assertions contained in the virtual ABox. Here we omit the formal definitions and the proof, but we note that it is straightforward to see that the specific result carries over to our modified definition of $P(Q, \mathcal{M})$. Also, the algorithm UnfoldDB is defined, which, given an UCQ Q over an ontology $\mathcal{O}$ with a mapping collection $\mathcal{M}$, translates the set of resultants returned by SLD-Derive($P(Q, \mathcal{M})$) into queries over the database instance D. Again, we omit the details and we note that in our case the resulted query will be a UCQ over S that has the form

$$Query(\vec{x}) \leftarrow Q_1(\vec{x}) \vee ... \vee Q_n(\vec{x}) \tag{1}$$

where each Q_i for $i = 1, ..., n$ is the translation given by UnfoldDB that corresponds to a resultant returned by SLD-Derive($P(Q, \mathcal{M})$), and it is an expression of the form

$$Q_i(\vec{f_i}(\vec{x_i})) \leftarrow$$
$$Aux_{i_1}(\vec{x_{i_1}}) \wedge ... \wedge Aux_{i_l}(\vec{x_{i_l}}) \wedge R_{i_{l+1}}(\vec{x_{i_{l+1}}}) \wedge ... \wedge R_{i_m}(\vec{x_{i_m}}) \tag{2}$$

where each $f_i^j \in \vec{f_i}$ is a function whose function name belongs in Λ and whose variable arguments are among the variables of $\vec{x_{i_1}}, ..., \vec{x_{i_m}}$, each Aux_{i_j} for $j = 1, ..., l$ corresponds to $body(m)$ for some $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$ and each R_{i_k} for $k = l + 1, ..., m$ is a relation name from the database schema. Note that on the original definition of UnfoldDB semantic query optimization (SQO) with respect to the database schema S is not performed. Nevertheless, in subsequent research, the role of SQO with respect to this context was proved crucial [27, 23]. In this work we consider that SQO, like self-join elimination, is performed in the result of UnfoldDB, that is in each Q_i for $i = 1, ..., n$ in (1). Furthermore, by overloading the definition of UnfoldDB, we consider a version of the function that takes as input an SLD-tree resulted from the application of the SLD-Derive($P(Q, \mathcal{M})$), and operates as described to produce a query that has the aforementioned form.

We now proceed with some definitions that will be used when we "fold back" the SLD-tree produced by SLD-Derive. For each edge e of the SLD-tree we define $source(e)$ to be the node at the beginning of e, $target(e)$ to be the node at the end of e, $TM(e)$ to be the predicate symbol of the atom selected by computation rule R at $source(e)$, $M(e)$ to be the clause (mapping assertion) used in the specific derivation, $sub(e)$ to be the substitution used in the specific derivation and $pos(e)$ to be the set of integers corresponding to the positions of atoms affected by the derivation in the right-hand side of the resultant in $target(e)$.

Let $m_1, ..., m_n$ be mapping assertions of the form $\phi_i \rightarrow \psi_i$ where no variable is repeated in ψ_i, for $i = 1, ..., n$. Also let θ be a unifier such that the atoms $\psi_1\theta, ..., \psi_n\theta$ are all equal (obviously the predicate symbol at the head of each assertion is the same). Then, the combined mapping of $m_1, ..., m_n$ is the following expression:

$\phi_1\theta \vee ... \vee \phi_n\theta \rightarrow \psi_1\theta$

If a variable z is repeated in some ψ_i, we modify ψ_i by keeping only the first occurrence and we replace all other occurrences with fresh variables $z_1, ..., z_k \in Var$. Then, we add the conditions $z = z_1, ..., z = z_n$ as conjuncts in the body of ψ_i.

Essentially, the combined mapping introduces a mapping assertion whose body is the union the input mappings, with the appropriate renaming. Two examples of combined mappings for the example mappings shown in Figure 2 are presented in Figures 5 and 6.

Proposition 1. Let T be an SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M})$, m_c be the combined mapping of mappings $m_1, ..., m_n \in \mathcal{M}$ and $\mathcal{M}_c = (\mathcal{M} \setminus \{m_1, ..., m_n\}) \cup \{m_c\}$. The semantics of $PE(Q, \mathcal{M})$ coincide with the semantics of $PE(Q, \mathcal{M}_c)$.

Proof. We need to show that for every tuple $\vec{t}$ of constants, $q(\vec{t})$ is true in $PE(Q, \mathcal{M})$ if and only if $q(\vec{t})$ is true in $PE(Q, \mathcal{M}_c)$, which follows directly from the construction of m_c. □

Let T be the tree resulted from SLD-Derive($P(Q, \mathcal{M})$) and e_0 an edge in T. Also, let $e_1, ..., e_n$ be edges in T with $source(e_0) = source(e_1) = ... = source(e_n)$ and $TM(e_0) = TM(e_1) = ... = TM(e_n)$ such that there exists a combined mapping $m_c : \phi_0\theta \vee ... \vee \phi_n\theta \rightarrow \psi_1\theta$, with $\theta = sub(e_0)$ and $TM(e_0)$ be equal to the predicate at the head of m_c. A fold of T into e_0 is the tree T_1 that is resulted from T by replacing in each descendant node of $target(e_0)$ (including $target(e_0)$) the atoms at positions $pos(e_0)$ with the

$$cm_1 : A_1(v_1^{m1}, v_2^{m1}) \vee A_2(v_1^{m1}, v_2^{m1}) \vee A_3(v_1^{m1}, v_2^{m1}, v_3^{m1''}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1}))$$
$$\theta = \{v_1^{m1'}/v_1^{m1}, v_1^{m1''}/v_1^{m1}, v_2^{m1'}/v_2^{m1}, v_2^{m1''}/v_2^{m1}\}$$

Figure 5: Combined Mapping for Mapping Assertions $m1$, $m1'$ and $m1''$

$$cm_2 : C_1(v_1^{m5}, v_2^{m5}) \vee C_2(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5}))$$
$$\theta = \{v_1^{m5'}/v_1^{m5}, v_2^{m5'}/v_2^{m5}\}$$

Figure 6: Combined Mapping for Mapping Assertions $m5$ and $m5'$

atom $\psi_1\theta$, and deleting all the sub-trees starting from $target(e_1), ..., target(e_n)$. Moreover, let $f_1, ..., f_m$ be all the edges in T (including e_0) such that $M(f_1) = ... = M(f_m) = M(e_0)$. Then, the fold of T based on m_c is the tree that is obtained if we sequentially apply the process of obtaining the fold of T into f_i for $i = 1, ..., m$ ensuring that for each f_k, f_l with k, l in $1, ...m$, if the depth of $target(f_k)$ in T is smaller than the depth of $target(f_l)$, then the fold of T into f_k is obtained after obtaining the fold of T into f_l.

Figure 7 shows the fold of the SLD-tree of example 2 based on the combined mapping from Figure 6, where Aux_{cm_2} is an auxiliary predicate used for mappings in $\mathcal{M} \setminus \mathcal{M}_{CQ}$ according to the construction of $P(Q, \mathcal{M})$, that correspond to combined mapping cm_2. Note that the same combined mapping is recognized and used in three different nodes of the initial tree.

Proposition 2. Let T be an SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M})$, m_c be the combined mapping of mappings $m_1, ..., m_n \in \mathcal{M}$ and $\mathcal{M}_c = (\mathcal{M} \setminus \{m_1, ..., m_n\}) \cup \{m_c\}$. The fold of T based on m_c is exactly the tree returned by SLD-Derive with input $P(Q, \mathcal{M}_c)$.

Proof. Let T_{fold} be the fold of T based on m_c and T_c be the SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M}_c)$. We need to show that T_{fold} and T_c consist of the same resultants. Clearly the two trees have the same root. Then, given a resultant $ans(\vec{x})\theta_0...\theta_k \leftarrow A_1(\vec{x_1}), ..., A_i(\vec{x_i}), ..., A_w(\vec{x_w})$ at depth k which is the same for the two trees, it is sufficient to show that the children of this resultant are the also the same for the two trees. Let $node_{T_c}$ and $node_{T_{fold}}$ be the nodes in T_c and T_{fold} respectively that contain the specific resultant. Suppose that A_i is the leftmost atom in the body of the resultant with predicate that belongs to the alphabet of $\mathcal{T}$ and will be chosen by the computation rule R. Also, for now, let us suppose that there is only one node $node_T$ in the initial tree T such that for every edge e in T with $TM(e)$ equal to the predicate symbol at the head of m_c, then $source(e) = node_T$. According to the construction of the fold of T into e_0, if $node_T$ is different from $node_{T_{fold}}$, then the children of $node_{T_{fold}}$ are the same with the children of $node_{T_c}$, as they are not affected by the combined mapping. If $node_T$ is equal to $node_{T_{fold}}$, then $node_T$ has n children affected by the combined mapping, plus a number of children not affected (possibly 0). The second kind of children are also children of $node_{T_c}$, whereas the first kind have been replaced in T_{fold} with the child $ans(\vec{x})\theta_0...\theta_k\theta_{k+1} \leftarrow A_1(\vec{x_1}), ..., Aux_c m(\vec{z}), ..., A_w(\vec{x_w})$, which is also a child of $node_{T_c}$, and these are the only children of both $node_{T_c}$ and $node_{T_{fold}}$. Now, if there are more nodes in T affected by the fold of T based on m_c, then from the construction of the fold, where descendant nodes are always modified prior to their predecessors, and from the fact that R chooses always the leftmost possible atom, it is straightforward to see that the result of the case where only one node is affected by the combined mapping is carried over to this case.

□

A direct consequence of Propositions 1 and 2 is that if we consider the SLD-tree tree T resulted from SLD-Derive with input $P(Q, \mathcal{M})$ and we apply the UnfoldDB algorithm on the resultants contained in the the fold of T based on the combined mapping m_c, then the SQL query that will be produced has exactly the same answers with the SQL

query produced by applying the UnfoldDB algorithm on the resultants of the original tree T. This gives us the ability to choose a sequence of folds, in order to obtain an equivalent translation that can be more efficient, by using a cost-based search in the initial SLD-tree, which we describe in detail in the Section 4.2.

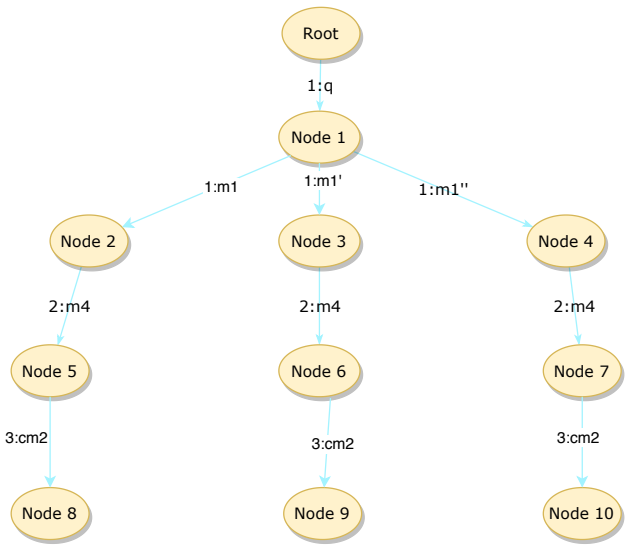
An issue that arises in these translations has to do with the way the combined mapping is treated. One way is to be treated as a regular mapping assertion, as the body of this mapping is simply a union query over the original mapping assertions. This union query will be computed as many times as the combined mapping is used in the produced query. Obviously a better choice would be to create a temporary table that holds the specific result as an intermediate result of the main query, in the same database connection. This is the solution that we follow in this work, as it also avoids the overheads of creating permanent materialized views in the database as in [26]. A second issue that has to be handled is the decision regarding which folds should be used, if any, for a specific query. As we will describe in the following section, the process of taking the specific decision heavily depends on the size of the SQL query, the size of each combined mapping in comparison to the size of the final SQL query and the number of duplicate answers contained in them.

4. Cost-Based Selection of Query Translation

In this section we consider a cost-based algorithm in order to choose a specific sequence of folds and obtain the SQL translation of the initial query. During this process we take into consideration the two kinds of redundant processing that we described in Section 1. Regarding the first kind (redundancy due to duplicates), we will employ a heuristic about early duplicate elimination of intermediate results during query evaluation that we first described in [1]. In order to describe the heuristic, we first consider a single subquery that has the form shown in formula (2) of Section 3. After that, in Section 4.2 we describe our algorithm operating on the complete query that has the form shown in formula (1). Our method relies on an estimation of the final result size of each union subquery. To obtain this estimation we should gather some statistics from the database in the form of data summarization for all the columns that can be possibly referenced from a query, that is all the columns in the SQL queries of some mapping assertion. As making an estimation for an arbitrary FOL query is an involved process, we make a distinction between assertions in $\mathcal{M}_{CQ}$ ($R_{i+l+1}, ..., R_{i+m}$ in formula 2) and assertions in $\mathcal{M} \setminus \mathcal{M}_{CQ}$ ($Aux_{i_1}, ..., Aux_{i_n}$ in formula 2). We consider that the latter are primitive tables as if they were virtual views, and we collect statistics only for the output columns, whereas the former are parsed and we collect statistics for all the referenced columns. We will refer to each conjunct in the right-hand side of (2) as an input table of query $Q_i(\vec{f_i}(\vec{x_i}))$.

Let q be a query as in (2) and $I_i(\vec{x_i})$ be an input table of q. The query $ans(\vec{x_{c_i}}) \leftarrow I_i(\vec{x_i})$, where $\vec{x_{c_i}}$ contains exactly the variables of $\vec{x_i}$ that appear at least two times in q, will be called the projection query of input table $I_i(\vec{x_i})$ from q. Additionally, let D be a database instance (which will be implied). Intuitively the projection query selects all the columns of an input table that are mentioned elsewhere in q. In Section 4.1, we decide if we will save each projection query as an intermediate result with respect to duplicates.

Analyzing External Tables. As we operate outside the RDBMS engine, in order to extract the needed information we should import all the corresponding data, which is clearly not practical. Luckily we have several other options. One such option is to only import a random sample and extract the needed information from that, as most database vendors support ordering the results by a random function. Another option is to obtain the data summarization directly from the RDBMS, if it provides a way to access this information. This option is likely to give the most accurate results, but it is highly dependant on the specificities of each database vendor. One third option is to build a simple single-bucket histogram for each column, by sending for execution queries that ask for the number of values, number of distinct values, minimum and maximum value. Simple histograms like this are known to give imprecise selectivity estimations for filter and join results of attributes that exhibit skewness [10], but on the other hand their construction and usage is faster in comparison to more elaborate kinds of histograms. For our experiments we have chosen the last option, as it is fast and simple and can be applied to any underlying RDBMS. This is a one-time offline process that needs to be done before query execution, similar to an analyze command in a database schema, as it only depends on the re-



$$Root : ans(x,y,z)\theta_0 \leftarrow ans(x,y,z)$$
$$\theta_0 = \{\}$$
$$Node1 : ans(x,y,z)\theta_0\theta_1 \leftarrow P_1(x,y), P_2(x,h(A)), P_3(y,z)$$
$$\theta_1 = \{\}$$
$$Node2 : ans(x,y,z)\theta_0\theta_1\theta_2 \leftarrow A_1(v_1^{m1}, v_2^{m1}), P_2(f(v_1^{m1}), h(A)), P_3(g(v_2^{m1}), z)$$
$$\theta_2 = \{x/f(v_1^{m1}), y/g(v_2^{m1})\}$$
$$Node3 : ans(x,y,z)\theta_0\theta_1\theta_3 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), P_2(f(v_1^{m1'}), h(A)), P_3(g(v_2^{m1'}), z)$$
$$\theta_3 = \{x/f(v_1^{m1'}), y/g(v_2^{m1'})\}$$
$$Node4 : ans(x,y,z)\theta_0\theta_1\theta_4 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), P_2(f(v_1^{m1''}), h(A)), P_3(g(v_2^{m1''}), z)$$
$$\theta_4 = \{x/f(v_1^{m1''}), y/g(v_2^{m1''})\}$$
$$Node5 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), P_3(g(v_2^{m1}), z)$$
$$\theta_5 = \{v_1^{m_4}/v_1^{m1}, v_3^{m_4}/A\}$$
$$Node6 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), P_3(g(v_2^{m1'}), z)$$
$$\theta_6 = \{v_1^{m_4}/v_1^{m1'}, v_3^{m_4}/A\}$$
$$Node7 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), P_3(g(v_2^{m1''}), z)$$
$$\theta_7 = \{v_1^{m_4}/v_1^{m1''}, v_3^{m_4}/A\}$$
$$Node8 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_8 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1}, v_2^{m5})$$
$$\theta_8 = \{v_1^{m5}/v_2^{m1}, z/k(v_2^{m5})\}$$
$$Node9 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{10} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1'}, v_2^{m5})$$
$$\theta_{10} = \{v_1^{m5}/v_2^{m1'}, z/k(v_2^{m5})\}$$
$$Node10 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{12} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1''}, v_2^{m5})$$
$$\theta_{12} = \{v_1^{m5}/v_2^{m1''}, z/k(v_2^{m5})\}$$

Figure 7: SLD Tree 2

lations occurring in mappings and data. Also, as it is crucial to have an accurate estimation of the number of duplicate answers that come from different mappings for the same predicate, we execute queries counting exactly the distinct number of answers for queries in bodies of mappings that can possibly formulate a combined mapping assertion. These mapping assertions can simply be identified offline as the subsets of mappings whose heads can be unified during the partial evaluation. Regarding duplicates coming from a single mapping, adopting the commonly used value independence assumption between the result attributes and the uniformity of values in an attribute [28], we estimate the distinct tuples of the relation to be the product of the distinct values of its attributes. In case this value is larger than the number of tuples in the relation, we assume that there are no duplicate tuples in the relation.

4.1. Early Duplicate Elimination of Intermediate Results

First, we define the duplicate-tuple ratio DTR_R of a relation instance R to be equal to $\frac{\sum_{t \in US_R} \mu(t)}{|US_R|}$. A relation instance with DTR equal to 1 will be called a duplicate-free relation instance. Now, let us suppose that we have a single SQL subquery coming from the unfolding step and we have to take the decision regarding a single input table (either "real" primitive table or virtual view) used in this subquery; we will take into consideration different union subqueries in Section 4.2. In this case, it may be advantageous to dictate the RDBMS to perform the duplicate elimination on projection query of the specific input table at the beginning of query execution, store the duplicate-free intermediate result in a temporary table and use it for the specific query. This can be done in several ways depending on the exact SQL dialect and capabilities of the underlying system. For example, one can use (non-recursive) common table expressions or temporary table definitions. Of course, the exact decisions as to when this should happen depend on several factors, including the exact query, the DTR of the projection query of the input table, the number of uses of the specific input table in the query, the choice to save the temporary table in disk or keep it in memory and several other factors that depend on the database physical design, database tuning parameters, the exact query execution plan and the evaluation methods chosen by the optimizer of the RDBMS. As mentioned, it is difficult for all these factors to be estimated outside the database engine. For this reason, in what follows, we propose to take this decision according to a heuristic that depends only on the size of the data and the DTR of the input table, whose estimation can be obtained using data summarization.

The main assumption that we make regarding duplicate elimination, states that the impact of an input table with DTR equal to a constant number n in the number of tuples of the final query result is proportional to n. As a result of this assumption, the selectivity of the query plays the most important role in duplicate elimination decisions. Intuitively, a query whose result size is much larger than the size of the intermediate result for which we examine the duplicate elimination option, it is expected to be faster to first perform the elimination, as each tuple of the intermediate result has as impact the creation of a large number of tuples in the final result. On the other hand, a query with few results is expected to be evaluated faster if duplicates are eliminated directly from the final result. In this case one would expect that each tuple of the intermediate result does not add that much to the total cost of the query in order to counterbalance the cost of a duplicate elimination, especially when expecting the optimizer to limit the sizes of intermediate query results as soon as possible.

A Heuristic Regarding Duplicate Elimination. Given a database instance D, a query q of the form (2) whose result over D is the relation instance Q and an input table $I_i(\vec{x_i})$ of q, perform duplicate elimination on input table $I_i(\vec{x_i})$ prior to execution of q if

$$Size_Q - \frac{Size_Q}{DTR_{Ans}} > \frac{Size_{Ans}}{DTR_{Ans}}$$

where relation instance Ans is the result of the projection query of $I_i(\vec{x_i})$ from q on D and $Size_Q$ and $Size_{Ans}$ are the estimated sizes (in bytes) of relation instances Q and Ans respectively. That is, duplicate elimination should be performed if it is expected that the reduction on the size of the final result will be bigger than the size of the intermediate result with duplicate elimination.

4.2. Cost-based Translation

In this section we present the algorithm *GetTranslation* (Algorithm 1), which, given a UCQ Q over an ontology $\mathcal{O}$ and a mapping collection $\mathcal{M}$

from O to a database instance D over a database schema S, it returns a SQL query over D and provides a set $CM_{temporary}$ of temporary views to be created. Each of these temporary views corresponds to a SQL query on the body of a combined mapping that exists in the SDL-tree produced by SLD-Derive($P(Q, \mathcal{M})$). In other words, the algorithm chooses a sequence of folds based on one of these combined mappings each time, that are performed repeatedly in a corresponding sequence of trees, starting from the initial SLD-tree. The $T_{current}$ variable holds the current tree at each point of execution. In each step, the fold that is expected to provide the largest gain is chosen, and this process is continued until no fold that provides gain exists. In this sense, the algorithm proceeds in a greedy way, in order to avoid examining all the combinations. The gain for each possible combined mapping is estimated based in the redundant processing that we avoid by materializing and using the specific mapping with respect to i) duplicate answers and ii) repeated operations even in the absence of duplicate answers.

Regarding duplicate answers, in correspondence with the observations made in Section 4.1, here the main factors that determine the behavior of the algorithm are the query selectivity and the size of the result of the SQL query in the body of each combined mapping. The difference here is that we consider the final query that is the result of UnfoldDB, instead of a single union subquery, and a combined mapping that contains many input mappings which can produce duplicate results between them, instead of a single input table of one subquery. Let cm be the combined mapping $\phi_1 \vee ... \vee \phi_n \rightarrow \psi$ in this context, for simplicity we will denote by $Size_{cm}$ and DTR_{cm} the size and DTR of the relation instance that is the result of executing the query $\phi_1 \vee ... \vee \phi_n$ over the database instance D, given that duplicate elimination is not performed. Computing and saving the combined mapping is expected to be more efficient, if the reduction on the size of the final SQL query will be bigger than the size of the temporary table resulting from the materialization of the combined mapping with duplicate elimination ($Size_{cm}/DTR_{cm}$). Using the quantity $Size_{SQL_{cm}}$ to denote the size of the result of the final SQL query when the combined mapping cm has been chosen for materialization with the duplicates eliminated, which is equal to $Size_{SQL_{current}}/DTR_{cm}$, we have that the result of UnfoldDB with input the fold of T into cm (SQL_{cm}) is preferred over the result of UnfoldDB with input T ($SQL_{current}$) if:

$$Size_{SQL_{current}} - Size_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \qquad (3)$$

Regarding repeated operations even in the absence of duplicate answers, as discussed in Section 1, in order to obtain an exact cost model we should be aware of the exact execution plan and the choice of access methods for each relation in order to estimate the amount of data read and written to disk for each CQ. As this is not viable for the OBDA system that operates outside the database engine, we base our estimation on the sizes of the input relations and the size of the result. Specifically, we consider that the smaller table in each CQ is fully scanned once, and all other tables are either probed using an index as many times as the number of final query results or are fully scanned once, depending on which of the two options has the lowest cost. In order to find the smaller table, table sizes in this context are compared by taking into consideration the filters that appear in each table in the CQ, that is tables are compared according to the size of each corresponding projection query. Also, as we do not want to take into consideration duplicates introduced from the combined mapping under consideration, for each input table that participates in the combined mapping, we take its size after we divide it by DTR_{cm}.

Let SQL be an SQL query of the form 1 that is the result of UnfoldDB, we will denote by RR_{SQL} the estimation for the size in bytes of redundant reads in the absence of duplicates as described. In other words, RR_{SQL} holds the sum of redundant reads for every disjunct (CQ) in the right-hand side of (1). Then, the result of UnfoldDB with input the fold of T into cm (SQL_{cm}) is preferred over the result of UnfoldDB with input T ($SQL_{current}$) if the estimated reduction in redundant reads from $SQL_{current}$ to SQL_{cm} is larger than the size of the temporary table resulting from the materialization of the combined mapping with duplicate elimination ($Size_{cm}/DTR_{cm}$):

$$RR_{SQL_{current}} - RR_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \qquad (4)$$

If we want to take both kinds of redundant processing into consideration concurrently, we simply have to add the left-hand side parts of (3) and (4):

$$Size_{SQL_{current}} - Size_{SQL_{cm}} + RR_{SQL_{current}} - RR_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \quad (5)$$

In Algorithm 1 we are considering the heuristic as a quantity giving the expected gain, with negative values meaning that we have loss instead of gain, as shown in Line 11 of the algorithm, since we want to compare the different options and choose the one that gives the biggest gain at each step. So the final formula used is:

$$Size_{SQL_{current}} - Size_{SQL_{cm}} + RR_{SQL_{current}} - RR_{SQL_{cm}} - \frac{Size_{cm}}{DTR_{cm}} \quad (6)$$

Regarding some implementation issues, we should note that we do not need to make selectivity estimation for all the results each time, but only for those that are affected by the combined mapping, that is, the disjuncts in the result of UnfoldDB that correspond to resultants in the SLD-tree which are descendants of nodes which use some of the input mappings of the combined mapping examined each time. As a matter of fact, we can modify the gain formula so that only these disjuncts are taken into consideration in the computation of $RR_{SQL_{current}}, RR_{SQL_{cm}}, SQL_{current}$ and SQL_{cm}.

5. Implementation and Experimental Evaluation

We have implemented our translation in an prototype extension of Ontop version 1.18.1. This version of Ontop normally uses the default unfolding method of [21] over the $\mathcal{T}$-Mappings in order to emulate H-complete ABoxes [22], as we mentioned in Section 1, and employs the tree-witness query rewriting [14] on such ABoxes. We follow the same architecture, using the tree-witness approach for query rewriting and we modify the unfolding step over the $\mathcal{T}$-Mappings as described here.

Newer versions of Ontop use a different query unfolding method that employs the notion of intermediate query (IQ) [30]. We discuss the relevance of our method to this in Section 6. For this reason, we compare our method with both the default translation based in partial evaluation of logic progrmams obtained by version 1.18.1, but also with the new

Algorithm 1: Translation Process

```
1  GetTranslation (M, Q, D);
   Input  : Mapping Collection M, Query Q,
            Database D
   Output: SQL query over D
2  CM_temporary = ∅;
   // The combined mappings that should be
      used as temporary tables
3  T_current = SLD-Derive(P(Q, M)); // The
      SLD-tree at each step. Initially equal to
      the result of SLD-Derive(P(Q, M))
4  SQL_current = UnfoldDB(T_current);
5  Add to CM_used all the combined mappings
      that exist in T_current;
6  MaxGain = 0;
7  do
8      foreach cm ∈ CM_used do
9          T_cm: the fold of T_current based on cm;
10         SQL_cm = UnfoldDB(T_cm);
11         Compute Gain from SQL_current to
             SQL_cm according to Formula 6 ;
12         if Gain > MaxGain then
13             MaxGain = Gain;
14             T_best = T_cm;
15             SQL_best = SQL_cm;
16             BestCm = cm;
17         end
18     end
19     if MaxGain > 0 then
20         SQL_current = SQL_best;
21         T_current = T_best;
22         Remove BestCm from CM_used;
23         Add BestCm to CM_temporary;
24     end
25 while MaxGain > 0;
26 return SQL_current;
```

translation method obtained from the latest Ontop versions 3.0.1 and 4.0.2. In general, version 3.0.1 outperforms version 4.0.2, so we only report times for version 3.0.1 here, but all the execution times for version 4.0.2 are also available along with all other material[2].

Our aim in this section is to perform an experimental comparison of our approach with other methods using well-known benchmarks. For this reason, we present experiments using the NPD and LUBM benchmarks in Section 5.1, comparing our approach with the translation performed by the two aforementioned Ontop versions. Then, in Section 5.2, we compare our approach with the JUCQ approach using the datasets and queries from [16] and, in Section 5.3, we study the performance of our method in comparison to the default translation, for different query characteristics. Finally, in order to obtain an empirical analysis of our heuristic regarding duplicate elimination, in Section 5.4 we perform an experimental evaluation using a micro benchmark with specific query fragments coming from queries used in the general evaluation.

5.1. Experiments with NPD and LUBM Benchmarks

We have performed an experimental evaluation of our techniques using the LUBM [9] and NPD [15] benchmarks, with the ontology and mappings that are publicly available at the Ontop repository on github[3] and with existential reasoning enabled. Both datasets were generated for scale 100.

The experiments in this section were carried out on a machine with an Intel Core i7-3770K processor with 8 cores and 32 GB of RAM running UBUNTU 18.04, using PostgreSQL version 11.3 as a backend. PostgreSQL was setup and tuned for usage in a machine with 32GB RAM. The schema and data in all systems were identical and all the proposed indexes were created. The database size was about 1.1 GB for LUBM and about 5.8 GB for NPD.

Queries and Mappings. For LUBM benchmark in total 84 mapping assertions were produced as $\mathcal{T}$-Mappings from Ontop. For LUBM we used the original 14 queries. For NPD we used a subset of 19 out of the original 30 queries: queries 1-12, 22-25 and 28-30, excluding the queries that use GROUP BY, as it is not supported by the used Ontop version, queries that contain OPTIONAL and queries with empty translation due to incompatible IRIs. To these queries we added four more, in order to showcase the advantage of duplicate elimination coming from a single mapping. The reason for this addition is that despite the fact that many mappings introduce duplicates, the existing queries are only using a small subset of the mappings that mostly avoid this problem. We believe that the four added queries are sensible and simple, yet their evaluation proved very hard. This showcases that the problem we are dealing with is also present in the NPD benchmark. These new queries are numbered 31 to 34 and presented in Appendix A. All SPARQL queries were executed using the DISTINCT modifier.

Overhead in Setup and Optimization. The time needed to gather all the necessary statistics and analyze tables prior to the first deployment of the system as described in Section 4 was 48 seconds for LUBM and 3 minutes and 10 seconds for NPD. Total optimization time for the 14 LUBM queries total time increased from 325 ms to 360 ms, whereas for the 23 NPD queries the increase was from 1115 ms to 1380 ms. The given times include the total time from parsing each SPARQL query to outputting the corresponding SQL query. The first time is the time needed by the original Ontop version 1.18.1, whereas the second time is the time needed by our modified version.

Results. For each query we used a timeout of 1000 seconds. For each setting, all queries were executed sequentially according to their numbering, after a full system reboot. The given times measure the total time needed for each query including the optimization time in Ontop, the execution time in the relational back-end and the time to obtain the results in Ontop. All the results were obtained, but they were not saved or processed otherwise. The combined mappings chosen by our method were materialized as temporary tables during execution in the same session as the main query and unique indexes were created on those tables. All times are in milliseconds. All results and the produced SQL queries, as well as all the necessary material to reproduce the experiments are available in the link given in the beginning of this section.

[2] http://cgi.di.uoa.gr/~dbilid/experiments-obda/

[3] https://github.com/ontop/iswc2014-benchmark/tree/master/LUBM and https://github.com/ontop/npd-benchmark

Query	v1 Default	v1 Opt.	v3	#Results
NPD 1	4899	5258	13696	1627744
NPD 2	4189	4142	5015	172751
NPD 3	1155	1119	1535	83737
NPD 4	20542	20899	27159	1627744
NPD 5	54	66	128	193
NPD 6	33234	23533	36128	1231564
NPD 7	1438	1377	1489	180
NPD 8	307	303	ERROR[1]	5974
NPD 9	2354	2222	1537	12750
NPD 10	4243	3649	3800	79512
NPD 11	86773	7650	8523	418056
NPD 12	122712	14376	16824	838430
NPD 22	6373	3247	8003	1113200
NPD 23	6565	3304	44340	763400
NPD 24	2437	498	ERROR[1]	147400
NPD 25	10055	9324	12106	1725400
NPD 28	32343	22815	167362	2141968
NPD 29	90271	17212	26400	419834
NPD 30	163276	26661	58143	705984
NPD 31	TIMEOUT	29771	54641	2979400
NPD 32	1085	318	746	8000
NPD 33	77139	19545	24509	148037
NPD 34	5443	3329	18678	486000
Avg.	30768[2]	9592	25274[2]	

[1] Error during unfolding
[2] Excluding timeouts and errors

Table 1: Results for NPD scale 100 (Times in ms)

Query	v1 Default	v1 Opt.	v3	#Results
LUBM 01	543	587	685	4
LUBM 02	1283	1272	1377	264
LUBM 03	129	87	101	6
LUBM 04	149	125	438	34
LUBM 05	69	98	71	719
LUBM 06	17086	8868	29419	1048532
LUBM 07	259	306	334	67
LUBM 08	393	301	1079	7790
LUBM 09	47126	33518	16539	27247
LUBM 10	16	16	13	4
LUBM 11	191	187	192	224
LUBM 12	132	134	245	15
LUBM 13	112	111	138	472
LUBM 14	3096	2826	4406	795970
Avg.	5042	3460	3931	

Table 2: Results for LUBM scale 100 (Times in ms)

Results are presented in Table 1 for NPD queries and in Table 2 for LUBM queries. Results in column v1 Default contains the execution times obtained by the Ontop version 1.18.1, column v1 Opt. contains the times obtained by the modified Ontop version according to our approach and column v3 contains the times obtained by Ontop version 3, the latest stable Ontop release. The average execution times for each case are also shown in the bottom of each table, excluding errors and timeouts. For the case of NPD queries, there was 1 timeout from v1 Default for query 31, and two errors during unfolding from Ontop v3. The exact error message for each error can be found at our result repository. According to the results, our approach outperforms on average both Ontop version 1.18.1 and version 3. For the NPD benchmark the decrease in average execution time obtained by our method is 69% and 62% in comparison to version 1.18.1 and version 3 respectively, while for the LUBM benchmark the decrease is 31% and 12% respectively. Also, with very few exceptions, our method outperforms the other two approaches on every single query.

5.2. Comparison with the JUCQ Approach

In this section we compare our method with the approach from [16]. As this implementation is not part of the Ontop release, we directly use the queries produced by this approach, which are available at the Ontop examples github repository [4]. For this reason, in all the experiments presented in this section we only report the time for executing the SQL queries in PostgreSQL, omitting the time for query unfolding. For measuring the execution times of the JUCQ approach, we used the scripts provided in the aforementioned github repository. As in the previous section, we also include the times obtained using the versions 1.18.1 and 3 of Ontop. The execution environment is the same as in the previous section.

We use the exact benchmark and queries that were also used in [16]. Specifically, we use the OBDA version of the Wisconsin benchmark [7], with the same ontology and mappings, for which we have created 24 instances of the base relational table, each one with 1 million tuples. This is the exact setting used in [16]. The results of the Wisconsin benchmark are presented in Table 3, where

[4] https://github.com/ontop/ontop-examples/tree/master/iswc-2017-cost/

there are two different query sets, one that contains queries consisting of 3 atoms, and the other with queries consisting of 4 atoms. Each query set contains 84 queries, and the average execution time for each approach is shown. Our approach outperforms all other translations, followed by the JUCQ approach, whereas the worst performance is obtained from the default translation of version 1, which is the only approach such that timeouts occur. One other observation has to do with the execution times for the UCQ (default translation of Ontop 1.18.1) and JUCQ translations reported in [16]. Specifically, our execution times for these two sets of approaches seem to be much better. For example, in their reported times, timeouts of 20 minutes occurred in every setting, and the average execution time for the JUCQ approach was 160 seconds for the 3 atoms query set, whereas in our experiments the corresponding time is only 30.5 seconds. These differences can possibly be attributed to different versions of the PostgreSQL database (they used version 9.6) and different tuning parameters of the database engine. Other than that, our findings are consistent with theirs. Specifically, we observed the largest improvement of JUCQ with respect to the default UCQ translation for queries with more mappings and redundancy. The behavior of our approach is similar, exhibiting large improvement for these queries in comparison to all other three approaches.

Finally, we use the same modified NPD queries NPD 6*, NPD 11*, NPD 12* and NPD 31* as in [16], executed over the scale 100 of the NPD benchmark. This is different from [16], where these queries were executed only over the original NPD dataset (scale 1). The results are presented in Table 4. Again our approach outperforms all other approaches. Also, regarding the JUCQ translation, the results here show a different situation in comparison to the Wisconsin benchmark, as it exhibits the worst performance and also a timeout occurs for query NPD 31*. The queries produced by the JUCQ approach seem in general more complicated from the ones produced from the other three approaches.

5.3. Performance gain

In this section, we study the performance gain of our optimized method over the default translation which is obtained by partial evaluation of logic programs and leads to generation of UCQs. Following the setup of [16], we use the Wisconsin benchmark,

Query Set	v1 Default	v1 Opt.	v3	JUCQ
3 atoms	73133	22589	53624	30528
4 atoms	223684[1]	30922	64048	43926

[1] Excluding 24 timeouts

Table 3: Average execution time (ms) for Wisconsin Benchmark (24 tables with 1 million tuples per table)

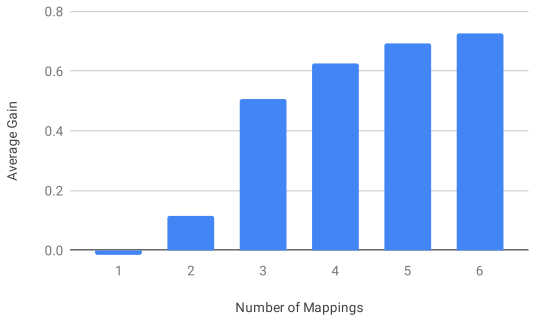


Figure 8: Performance gain for varying number of mappings per predicate

generating 24 tables with 1 million tuples per table, executing 84 queries with 3 atoms each, with a varying number of mappings used for each query (from 1 to 6) and we compute the performance gain using the formula 1−(Opt. Time/Default Time). In Figure 8 we present results for each number of mappings per predicate. The figure presents the average gain for all the queries per case (1 to 6 mappings). As expected, when there is only 1 mapping per predicate, our method does not generate any temporary table, and as a result, it performs roughly the same as the default translation. Starting from two predicates, our methods begins to outperform the default translation, reaching an average gain of more than 0.7.

In Figure 9, we present a scatter chart with the performance gain with respect to the number of results for the 84 queries of the benchmark. As shown, the queries are partitioned in visually distinct groups with respect to the number of their results. The effect of query selectivity is evident in this chart, with our method becoming increasingly efficient compared to the default, as the number of results grows larger, achieving a gain of 0.75 for the queries with about 1.7 million results. On the contrary, for the first group of queries, with low number of results, we cannot see a consistent behavior in comparison with the default.

Query	v1 Default	v1 Opt.	v3	JUCQ [16]	#Results
NPD 6*	91083	21700	132787	295445	2150854
NPD 11*	169833	9189	20070	204426	734214
NPD 12*	74001	5415	11471	14699	734214
NPD 31*	224925	18201	3980	ERROR[1]	1718
AVG	139960	13626	42077	171523[2]	

[1] Error during execution after 221 seconds
[2] Excluding Errors

Table 4: Results for NPD queries from [16] (scale 100-Times in ms)

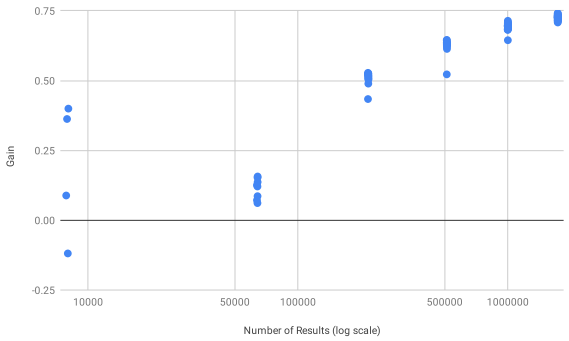


Figure 9: Performance gain with respect to number of results

5.4. Evaluating the Duplicate Elimination Heuristic

In this section we present experimental justification for the use of our heuristic regarding duplicate elimination. For this purpose, we have chosen four query fragments from the LUBM benchmark and four from NPD, such that duplicate elimination is applicable on them, as it was found during the previously described experiments. The experiments of this section were carried out on a machine with an Intel Core i7-3770K processor with 8 cores and 16 GB of RAM running UBUNTU 16.04. As our intention was to examine how our optimizations perform in different underlying systems, we used four different back-ends: PostgreSQL (version 9.3), MySQL (version 5.7) and two of the most widely used proprietary RDBMSs, which due to their license we will call System I and System X. All systems were setup and tuned for usage in a machine with 16GB RAM.

Each query fragment consists of a single select-from-where subquery. The fragments were chosen such that they have varying characteristics regarding the execution time, the number of results and the DTR of the mapping assertion under consideration. In order to test these queries with different selectivities, we applied to them extra filters. As LUBM100 contains information about exactly 100 universities, we used a filter on the university ID attribute in direct correspondence to the percentage of selectivity, whereas for NPD we used different filters for each fragment. We used filters that result in selectivity percentage of 1, 5, 10, 30 and 60, resulting in a total of 40 queries per system. We executed each of these 40 queries with and without duplicate elimination performed, resulting in a total of 240 runs for all systems. The results were obtained with warm caches.

In the upper part of Table 5 (one-time) we present the total execution times for these queries per system, depending on the duplicate elimination strategy. The titles of the first three columns are self-explanatory. The fifth column gives the total time, if always the best strategy was chosen for each system. The fourth column gives the best time, if for each query and each selectivity, the best common strategy was chosen for all systems. This way, the difference between the fourth and fifth column can give an indication of how similar the behaviors of the systems are, whereas comparison of third and fourth columns can give a measure of how well our heuristic takes advantage of this common behavior.

One can observe that the strategy of always performing duplicate elimination is much better than never performing, and that even the strategy of always choosing the best approach is not extremely better. The reason for this result is that for queries with low selectivity, the execution time is much larger and dominates the total time. For these queries, performing duplicate elimination is preferable and sometimes gives up to two orders of magnitude better results. In order to simulate a query mix such that low selectivity queries do not dominate execution time, we also computed results where we give very selective queries a weight, such

that queries with 1% selectivity have been executed 60 times, queries with 5% selectivity have been executed 12 times, etc. We present the total execution time under this setting in the lower part of Table 5. As before, exact times and queries are available at the same location [5].

6. Related Work and Conclusions

Regarding related work, the research of Lanti et al. [16] constitutes the most relevant to ours, as it also deals with cost-based translation. The authors extend the cover-based translation of Bursztyn et al. [3], in order to take into consideration the mappings to arbitrary relational schemas. The authors analyze the database as a preprocessing step, in order to extract useful statistics, such as the cardinality of join results between queries in bodies of mapping assertions whose heads can be joined. Using these statistics, the authors can obtain accurate selectivity estimations for the produced queries. Unfortunately, despite the accurate selectivity estimations, the cost model used to compare the different cover-based reformulations is not realistic, as it assumes that all joins in a CQ are performed using hash joins, which is highly unlikely, and also it is assumed that every input relation is completely scanned. Also, the join order is not taken into consideration at all, something that can have a huge impact in the cost of the query. As we have discussed, this is an inherent problem of a system that operates outside the database engine. The difference with our method is that we use heuristics that apply to different execution plans and database engines, and also, at each step of our method, we compare highly relevant queries, where apart from the relations affected by the combined mapping under consideration, all other input relations and joins between them are the same, such that query selectivity plays the most important role in our decision. Also, we avoid running the query translation process multiple times, whereas in [16] for each different query cover, the rewriting, unfolding and estimation process has to be performed independently. Finally, the authors only consider mappings whose the body is always a CQ over the relational schema.

Since version 3, the Ontop system has departed from the usage of partial evaluation of logic programs for query unfolding. Specifically, it now relies on a query representation which is called intermediate query [30], in order to represent both SPARQL and SQL queries, facilitating the translation of SPARQL query operators like OPTIONAL [29] and GROUP BY. Instead, in this work we concentrate only on CQs over the ontology. We have experimentally shown that our method performs better on average for CQs in comparison with the latest Ontop versions. We believe that it is an interesting topic for future research to also apply cost-based methods to other operators present in SPARQL, possibly combining our results with the line of research carried out in [30, 29].

The work presented by Sequeda et al. [26] is also relevant, as it uses a cost model in order to materialize specific views prior to query execution. This solution in many cases provides efficient query execution, but incurs expensive preprocessing and also, using materialized views in the database increases the database maintenance load, especially for frequently updated tables, as well as the the database size. Also, it is not in line with the overall OBDA approach of providing the end user with access to several underlying data sources, without the need to modify data, and on a practical level, such access may not be even possible. In contrast, we compute specific temporary views during query execution, when we estimate that this will result in lower execution cost, without affecting the original database schema.

Jacques et al. [12] adopt a logic which enables them to avoid mappings when using an object-relational back-end and a combination of data completion and query rewriting. During this process primary keys are used for object identification, removing the need for duplicate elimination. Also, the authors use disjointness axioms in the ontology to further remove the need of duplicate elimination between unions. Gottlob et al. [8] present query rewriting and optimization techniques that eliminate redundant atoms during the application of a resolution based algorithm. To do so, they employ a method that takes into consideration the tuple-generating dependencies (TGDs) of the ontological language they consider, which unlike the DL-Lite languages, considers atoms of arbitrary arity, thus it is conceptually closer to the relational model and does not need separate mappings, so a separate unfolding phase is not needed.

We have identified redundant processing as a bottleneck in OBDA query processing and we have proposed solutions to overcome this problem. We

[5] http://cgi.di.uoa.gr/~dbilid/experiments-obda/

	System	Always	Never	Heuristic	Best (common)	Best(Separate)
one-time	PostgreSQL	13345	168785	12854	12638	12353
	MySQL	281598	-	281685	279522	279265
	SystemI	10733	143616	9906	9693	9502
	SystemX	20558	27479	8588	8803	7280
query-mix	PostgreSQL	167116	618328	144984	146406	143191
	MySQL	1129311	-	1066499	1056659	1056145
	SystemI	135790	520408	102724	101984	99989
	SystemX	167761	220408	93660	90557	83045

Table 5: Query Results for Different Duplicate Elimination Strategies (Times in sec.)

believe that using cost-based planning is a prominent direction towards OBDA query optimization, that has not been fully explored yet. In future work, we plan to incorporate decisions about physical database design by analyzing the mapping assertions. One more direction regarding future research has to do with duplicate elimination in case the OBDA system is equipped with query processing capabilities, in other words when it acts as a mediator. In this setting, along with decisions regarding which query fragments should be evaluated in external databases, one should decide when duplicate elimination should be "pushed" to endpoints or performed by the OBDA processing engine during data import.

Acknowledgments

The present work was co-funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 825258, and by the Seventh Framework Program (FP7) of the European Commission under Grant Agreement 318338.

References

[1] Bilidas, D., Koubarakis, M.: Efficient duplicate elimination in SPARQL to SQL translation. In: Description Logics (2018)

[2] Bitton, D., DeWitt, D.J.: Duplicate record elimination in large data files. ACM Transactions on database systems (TODS) 8(2), 255–265 (1983)

[3] Bursztyn, D., Goasdoué, F., Manolescu, I.: Teaching an RDBMS about ontological constraints. Proceedings of the VLDB Endowment 9(12), 1161–1172 (2016)

[4] Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering sparql queries over relational databases. Semantic Web 8(3), 471–487 (2017)

[5] Chaudhuri, S., Vardi, M.Y.: Optimization of real conjunctive queries. In: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. pp. 59–70. ACM (1993)

[6] Chortaras, A., Trivela, D., Stamou, G.B.: Optimized query rewriting for OWL 2 QL. In: CADE. vol. 11, pp. 192–206. Springer (2011)

[7] DeWitt, D.J.: The wisconsin benchmark: Past, present, and future. In: Gray, J. (ed.) The Benchmark Handbook for Database and Transaction Systems (2nd Edition). Morgan Kaufmann (1993)

[8] Gottlob, G., Orsi, G., Pieris, A.: Query rewriting and optimization for ontological databases. ACM Transactions on Database Systems (TODS) 39(3), 25 (2014)

[9] Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2), 158–182 (2005)

[10] Ioannidis, Y.: The history of histograms (abridged). In: Proceedings of the 29th international conference on Very large data bases-Volume 29. pp. 19–30. VLDB Endowment (2003)

[11] Ives, Z.G., Florescu, D., Friedman, M., Levy, A., Weld, D.S.: An adaptive query execution system for data integration. ACM SIGMOD Record 28(2), 299–310 (1999)

[12] Jacques, J.S., Toman, D., Weddell, G.E.: Object-relational queries over $CFD^{\forall-}_{nc}$ knowledge bases: OBDA for the SQL-literate. In: Description Logics (2016)

[13] Kharlamov, E., Hovland, D., Jiménez-Ruiz, E., Lanti, D., Lie, H., Pinkel, C., Rezk, M., Skjæveland, M.G., Thorstensen, E., Xiao, G., Zheleznyakov, D., Horrocks, I.: Ontology based access to exploration data at Statoil. In: International Semantic Web Conference. pp. 93–112. Springer (2015)

[14] Kikot, S., Kontchakov, R., Zakharyaschev, M.: Conjunctive query answering with OWL 2 QL. In: Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning (2012)

[15] Lanti, D., Rezk, M., Xiao, G., Calvanese, D.: The NPD benchmark: Reality check for OBDA systems. In: Proc. of the 18th Int. Conf. on Extending Database Technology (EDBT) (2015)

[16] Lanti, D., Xiao, G., Calvanese, D.: Cost-driven ontology-based data access. In: The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10587, pp. 452–470. Springer (2017)

[17] Lloyd, J.W.: Foundations of logic programming.

Springer Science & Business Media (2012)
[18] Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. The Journal of Logic Programming 11(3-4), 217–242 (1991)
[19] Park, J., Segev, A.: Using common subexpressions to optimize multiple queries. In: Data Engineering, 1988. Proceedings. Fourth International Conference on. pp. 311–319. IEEE (1988)
[20] Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for owl 2. The Semantic Web-ISWC 2009 pp. 489–504 (2009)
[21] Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. In: Journal on data semantics, pp. 133–173. Springer (2008)
[22] Rodriguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-based data access: Ontop of databases. In: International Semantic Web Conference. pp. 558–573. Springer (2013)
[23] Rodríguez-Muro, M., Rezk, M.: Efficient SPARQL-to-SQL with R2RML mappings. Web Semantics: Science, Services and Agents on the World Wide Web 33, 141–169 (2015)
[24] Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. ACM SIGMOD Record 29(2), 249–260 (2000)
[25] Sellis, T.K.: Multiple-query optimization. ACM Transactions on Database Systems (TODS) 13(1), 23–52 (1988)
[26] Sequeda, J.F., Arenas, M., Miranker, D.P.: OBDA: query rewriting or materialization? in practice, both! In: International Semantic Web Conference. pp. 535–551. Springer (2014)
[27] Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL execution on relational data. Web Semantics: Science, Services and Agents on the World Wide Web 22, 19–39 (2013)
[28] Swami, A., Schiefer, K.B.: On the estimation of join result sizes. In: International Conference on Extending Database Technology. pp. 287–300. Springer (1994)
[29] Xiao, G., Kontchakov, R., Cogrel, B., Calvanese, D., Botoeva, E.: Efficient handling of sparql optional for obda. In: International Semantic Web Conference. pp. 354–373. Springer (2018)
[30] Xiao, G., Lanti, D., Kontchakov, R., Komla-Ebri, S., Güzel-Kalaycı, E., Ding, L., Corman, J., Cogrel, B., Calvanese, D., Botoeva, E.: The virtual knowledge graph system Ontop. ISWC 2020 - 19th International Semantic Web Conference (2020)

Appendix A. NPD Queries 31-34

```
SELECT DISTINCT ?q ?u
WHERE {
?q :inLithostratigraphicUnit ?u .
?u rdf:type :LithostratigraphicUnit .
}
```

Listing 1: Query NPD 31

```
SELECT DISTINCT ?quadrant ?name
WHERE {
?quadrant rdf:type :Quadrant .
?quadrant :name ?name .
}
```

Listing 2: Query NPD 32

```
SELECT DISTINCT ?unit ?era
WHERE {
?unit :geochronologicEra ?era .
?unit rdf:type :LithostratigraphicUnit .
}
```

Listing 3: Query NPD 33

```
SELECT DISTINCT ?wellbore ?discovery ?year
WHERE {
?wellbore rdf:type :Wellbore .
?wellbore :wellboreForDiscovery ?discovery .
?discovery :discoveryYear ?year
}
```

Listing 4: Query NPD 34