Efficient Duplicate Elimination in SPARQL to SQL Translation

Dimitris Bilidas and Manolis Koubarakis

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens, Greece
{d.bilidas,koubarak}@di.uoa.gr

Abstract. Redundant processing is a key problem in SPARQL to SQL query translation in *Ontology Based Data Access* (OBDA) systems. Many optimizations that aim to minimize this problem have been proposed and implemented. However, a large number of redundant duplicate answers are still generated in many practical settings, and this is a factor that impacts query execution. In this work we identify specific query traits that lead to duplicate introduction and we track down the mappings that are responsible for this behavior. Through experimental evaluation using the OBDA system Ontop, we exhibit the benefits of early duplicate elimination and show how to incorporate this technique into query optimization decisions.

1 Introduction

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, providing 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*, where the initial query is transformed in order to take into consideration the ontology axioms, and *query unfolding* where 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 ontology is linked through mappings to data stored in a *relational database management system* (RDBMS) providing the user with access to a virtual RDF graph. The original query is expressed over this virtual RDF graph in the SPARQL query language, and the result of rewriting and unfolding is a SQL query. As an example consider the relational tables from Figures 1a and 1b and the mappings from Figure 1e. In these mappings *hasDirector* and *hasActor* are properties defined in the ontology, whereas f, g and h are functions roughly corresponding to string concatenation. These functions are responsible for constructing an object that acts as an ontology instance out of values occurring in the database. In our setting, they construct an RDF term represented by an IRI.

If an RDF graph is lean (i.e., if it has no instance which is a proper subgraph of itself), evaluation of basic graph pattern on this graph can never result in duplicate answers [16]. However, duplicates in SPARQL query evaluation can be introduced from the operations of projection and union. Therefore, it is crucial that SQL query fragments resulting from the translation of basic graph patterns are also duplicate free when evaluated in a source database. Note also that, as there is no restriction regarding the multiplicity of the results of SQL queries used in a mapping, duplicates may be introduced even during evaluation of a single triple pattern.

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 input tables, in a query that involves several joins. 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 [10] 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 what follows we briefly identify reasons for this behavior.

Given that relational schemas are normalized, duplicates usually show up in SQL queries due to projections. As long as an OBDA mapping that generates virtual triples uses a column or combination of columns whose values are unique, e.g. a primary key of a table as the subject or object of each triple, then all triples coming from this mapping will be distinct. However, a denormalized schema or mappings that do not use unique columns can lead to duplicate introduction even from a single triple pattern. One more case in which duplicates are introduced has to do with the use of property *rdf*:*type*. Mappings defining *rdf*:*type* property should also be duplicate free, but as OBDA systems employ reasoning with respect to domain and range of properties, part of the mapping that defines a property could be used for evaluating a triple pattern with *rdf*:*type*. Then the other part of the mapping is projected out, something that can lead to a large number of duplicates. This is illustrated in Example 1.

Example 1. Consider the mappings from Figure 1e and also that the ontology defines the range of property *hasDirector* to be the class *Director*. Now consider a triple pattern ?*x* *rdf*:*type* *Director* that asks for the entities that belong to the class *Director*. One way to obtain such entities is to use the entities participating as objects to the property *hasDirector*. As we are not interested in the subjects of the generated triples, we are projecting out the *title* column, adding only a condition that this column should not be null. The resulting SQL to be sent for evaluation is shown in Figure 1c (where || represents the string concatenation operator in SQL)

In the example above, the number of duplicate results is equal to the average number of movies that correspond to each director in the *movies* table. This can become a heavy burden for query evaluation, especially when the query involves several joins, as all these duplicate values need to be joined with other tables, that may as well contain redundant values, and as a result, the overall cost can be increased by orders of magnitude, depending on the exact number of duplicates. A second issue is that, as the final answers need to be distinct, a duplicate elimination has to be performed on the final query result that consists of tuples of RDF terms instead of database values, making the relational engine perform this over (usually large) string values. As a typical query translation produces a UNION SQL query, results from different subqueries have to be merged and deduplicated.

In this work we present efficient solutions to the described problems, 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. We start by providing some preliminaries regarding ontologies, mappings and relational databases (Section 2), including a condition that guarantees distinct results for specific queries, by taking into consideration functional database dependencies. We then proceed to describe a process, that given an initial mapping collection, identifies the (possibly modified) mapping assertions which are responsible for introduction of duplicates (Section 3). Deduplicated results of these assertions can be materialized offline and replace the original assertions during query execution. In case materialized views are not a viable option, for example because of read-only access to data or for efficiency reasons, we proceed to deal with the problem of duplicates during query execution.

In Section 4, we describe how a structural optimization of pushing duplicate elimination before IRI construction can be applied and obtain an equivalent query with the one produced during unfolding. Then, in Section 5 we propose a heuristic that can help an OBDA system acting outside the relational engine to take decisions regarding the duplicate elimination of intermediate results that have been produced by the previously identified mapping assertions. This heuristic uses only information regarding the size of source relations and an estimation of the final size of the query, using easily obtained data summarization information from the underlying relational engine. In Section 6 we propose an improvement to self-join elimination methods currently employed by OBDA systems, relevant to redundancy due to duplicates, as it is applied in cases of denormalized relational schemas, or mappings that do not use primary keys, which both lead to introduction of duplicates.

Having implemented our optimizations in the state of the art OBDA system Ontop [19], in Section 7 we present their impact on NPD and LUBM benchmarks

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

title1	director1
title2	director1
title1	director2

(a) Table movies(title, director)

title1	actor1
title1	actor2
title1	actor3
title2	actor1
title2	actor4
title2	actor5

(b) Table actors(movie_title, actor)

```
SELECT 'IRI' || director
FROM movies
WHERE title IS NOT NULL
```

(c) SQL query

```
SELECT * WHERE {
    ?x a Movie .
    ?x hasDirector ?y .
    ?x hasActor ?z . }
```

(d) SPARQL query

$$movies(t, d) \rightarrow hasDirector(f(t), g(d))$$
$$movies(t, d), actors(t, a) \rightarrow hasActor(f(t), h(a))$$

(e) Example Mappings

Fig. 1: Example Database

using three different relational back-ends. Specifically we show that duplicates are present in many mappings from both benchmarks and that specific queries are heavily impacted by this fact, leading to evaluation times of more than 20 minutes, which in some cases can be reduced to a few seconds. Also we show that, excluding those queries that give a timeout of more than 20 minutes in the unoptimized setting, the overall improvement for the rest of the queries can be more than 60%. We also show that our self-join optimization affects several queries from both benchmarks and that pushing duplicate elimination before IRI construction, apart from reducing the overall execution time, has also the effect of obtaining the answers for many queries in a pipelined fashion, leading specifically to large improvement in the time of obtaining the first answers. Finally, we evaluate our heuristic and show that its usage is justified and that for query mixes from the two 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. In Section 8 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 DB_{Const} of database constants and $\mathcal{O}_{Const}$ of ontology constants.

As in [18], 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. These functions roughly correspond to IRI templates specified in the R2RML[2] language.

2.1 Databases.

We start by giving definitions for database instances and queries over them, following the bag semantics from [4].

A *bag* B is a pair (A, m), where A is a set called the underlying set of B and m is a function from elements of A to the positive integers, which gives the multiplicities of elements of A in B. The *underlying set* of a bag B will be denoted as US_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 [4]. 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: $Query(\boldsymbol{x}) \leftarrow \alpha$, where α is a first order expression containing predicates from Σ_R, which are among the relations that belong to S, $Query \in \Sigma_R$, $Query \notin S$ and $\boldsymbol{x}$ is a vector of constants from DB_{Const} and variables from Var that appear in α.

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

Let q be the SQL query $Query(\boldsymbol{x}) \leftarrow \alpha$, we will denote as $\prod_i(q)$ the query resulting from the projection of the i-th answer term of q, that is the query: $Query(x_i) \leftarrow \alpha$

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

[2] https://www.w3.org/TR/r2rml/

variable in Q. We denote by $\theta(X)$ the constant in DB_{Const} to which θ maps X and we denote by $\theta(R_i(\boldsymbol{x_i}))$ the ground atom to which $R_i(\boldsymbol{x_i})$ is mapped.

Let m_i denote the multiplicities $m(\theta(R_i(\boldsymbol{x_i}))), i = 1, ..., n$. The *result* due to θ of a conjunctive query Q over D is the atom $(\theta(\boldsymbol{x}), m_Q)$ with the multiplicity $m_Q = m_1 m_2 \cdots m_n$.

The *result* of a conjunctive query Q over a database instance D denoted $Q(D)$ is given by $\cup_\theta r_\theta$, where θ is any assignment mapping of Q into D and r_θ is the result due to θ.

2.3 Ontology and Mappings.

A $TBox$ is a finite set of ontology axioms.

An $ABox$ is a finite set of membership assertions $A(c)$ or role filling assertions $P(c, c')$, where $c, c' \in \mathcal{O}_{Const}$ and $A, P \in \Sigma_O$ denote a concept name and role name respectively.

A DL ontology $\mathcal{O}$ is a pair $\langle \mathcal{T}, \mathcal{A} \rangle$ where $\mathcal{T}$ is a $TBox$ and $\mathcal{A}$ an $ABox$.

A *mapping assertion* m from a source schema S to a $TBox$ $\mathcal{T}$ has the form: $\phi(\boldsymbol{x}) \rightarrow \psi_1 \wedge ... \wedge \psi_n$, where $\phi(\boldsymbol{x})$ will be denoted *body(m)* and it is a SQL query over a database instance D, each ψ_i has the form $P_i(cc_i^1(\boldsymbol{x_i^1}), cc_i^2(\boldsymbol{x_i^2}))$ or $C_i(cc_i^1(\boldsymbol{x_i^1}))$ with P_i (respectively C_i) $\in \Sigma_O$ a property (respectively class) name, and each $cc_i^j \in \Lambda$ is a function with arity equal to the length of $\boldsymbol{x_i^j}$ and range a subset of $\mathcal{O}_{Const}$. The conjunction in the right hand side will be denoted *head(m)*.

In this paper we consider *global-as-view* (GAV) mappings, where all variables in $\psi_1 \wedge ... \wedge \psi_n$ also appear in $\boldsymbol{x}$. In this setting, M can be transformed into a set of equivalent mapping assertions where the head of each assertion consists of a single atom [18]. In what follows we assume that the head of every mapping assertion consists of a single atom. A *mapping* $\mathcal{M}$ is a finite set of such mapping assertions.

Let $\mathcal{M}$ be a mapping, 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 $\mathcal{O}$ as an expression of the form: $Query(\boldsymbol{x}) \leftarrow P_1(\boldsymbol{x_1}) \wedge ... \wedge P_n(\boldsymbol{x_n})$ where $\boldsymbol{x_1}, ..., \boldsymbol{x_n}$ are vectors of constants from $\mathcal{O}_{Const}$ and variables from Var, $\boldsymbol{x}$ is a vector of constants from $\mathcal{O}_{Const}$ and variables from Var that appear in $\boldsymbol{x_1}, ..., \boldsymbol{x_n}$, and $P_1, ..., P_n \in \Sigma_O$ are ontology predicates that appear in $\mathcal{O}$.

A *union of conjunctive queries* UCQ over an ontology $\mathcal{O}$ is an expression of the form $Query(\boldsymbol{x}) \leftarrow CQ_1(\boldsymbol{x}) \vee ... \vee CQ_n(\boldsymbol{x})$, where each CQ_i for $i = 1...n$ is an expression of the form $P_1^i(\boldsymbol{x_1^i}) \wedge ... \wedge P_n^i(\boldsymbol{x_n^i})$ as in the previous definition.

2.4 Primary Keys and Duplicate Free Results.

The *duplicate-tuple ratio* DTR_R of a relation instance R is equal to $\frac{\sum_{a \in US_R} m(a)}{|US_R|}$. A relation instance with DTR equal to 1, will be called a *duplicate-free* relation instance.

As a first step, we want to take advantage of primary key constraints that hold in the source schema, so that we will avoid estimating DTR for relations which can be deduced to be duplicate-free. It is known that the interaction of key dependencies and functional dependencies is different under set and bag semantics [12]. In what follows we are only interested in key dependencies, since only these can be proved useful in order to decide if a result of a conjunctive query is duplicate-free, and also they correspond to the primary key declarations of SQL.

Let R be a relation of arity k in a source schema S and X, Y subsets of $\{1, 2, ..., k\}$ with $X \cap Y \subseteq \varnothing$ and $X \cup Y = \{1, 2, ..., k\}$. For an atom $R(\boldsymbol{a})$ the *projection* of terms using X is the set of terms from $\boldsymbol{a}$ whose subscripts correspond to the elements of X. We will denote this set as $\pi_X(R(\boldsymbol{a}))$. We say that a *key dependency* $X \rightarrow Y$ holds for R in a database instance D of S, if there are no distinct atoms $R(\boldsymbol{a})$ and $R(\boldsymbol{b})$ in D such that $\pi_X(R(\boldsymbol{a})) = \pi_X(R(\boldsymbol{b}))$ and for every atom $R(\boldsymbol{a})$ in D, $m(R(\boldsymbol{a})) = 1$. It is straightforward to see that any instance of a relation for which there is a primary key, is duplicate-free.

Later on, in Section 3, when we will want to see if the result of a mapping assertion is duplicate-free, we will use the following important proposition of [15]:

Proposition 1. *Let Q be a conjunctive query over a source schema S. Then, if for each $R_i(\boldsymbol{u_i})$ in the body of Q there is a key dependency $X \rightarrow Y$ in S and for each element $u_{i_j} \in \pi_X(R(\boldsymbol{u_i}))$, either $u_{i_j} \in DB_{Const}$ or u_{i_j} appears in $Query(\boldsymbol{x})$, then the result of $Query(\boldsymbol{x})$ is a duplicate-free relation instance on any database instance for S.*

2.5 Query Rewriting and Unfolding

As we mentioned in Section 1, query answering in OBDA consists of query rewriting and query unfolding. During query rewriting, an initial CQ 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 the ABox of the initial ontology alone (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 [18]. We omit details, as in this work we mainly consider the result query of this process. We just need to note that several methods exist for query rewriting over OWL 2 QL ontologies. In this work we consider that the result of the query rewriting step is a UCQ over the ontology, as it is the case for several rewriting methods [18, 11, 5, 17]. We discuss applicability on rewriting methods producing different forms of queries in Section 8.

Regarding query unfolding with respect to a Mapping $\mathcal{M}$, a method based on partial Datalog evaluation with functional terms is presented in [18]. As before, we omit a detailed description and note that the result of this process is a query over the relational schema that has the form

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

where each Q_i for $i = 1...n$ is an expression of the form

$Q_i(\boldsymbol{f_i}(\boldsymbol{x_i})) \leftarrow Aux_1(\boldsymbol{x_1^i}) \wedge ... \wedge Aux_l(\boldsymbol{x_l^i})$

where each $f_i^j \in \boldsymbol{f_i}$ is a function whose function name belongs in Λ and whose variable arguments are among the variables of $\boldsymbol{x_1^i}, ..., \boldsymbol{x_l^i}$ and each Aux_j for $j = 1...l$ corresponds to $body(m)$ for some $m \in \mathcal{M}$.

3 Offline Duplicate Elimination With Materialized Views

One solution to the problem of duplicates, is to track down the mapping assertions which are responsible for duplicates, create materialized views with the distinct results, possibly with indexes, and then use these views instead of the original assertions during query unfolding. It is reasonable to expect that this solution will give the best performance during query execution, but on the other hand this 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, this solution is not in line with the overall 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.

Nevertheless, even in the case where one chooses to use materialized views, it is not straightforward exactly which of the mapping assertions should be chosen. In the rest of this section we describe a process to find the exact assertions for this setting, whereas in the following sections we consider the case where no materialization happens and all processing needs to be done during query execution.

Given a mapping $\mathcal{M}$ and a database instance D over a schema S, a straightforward solution is to materialize all $m \in \mathcal{M}$ such that $DTR_{head(m)(D)} > 1$, but as the query produced after rewriting takes into consideration the ontology axioms, implied assertions may be used, such that a specific variable has been projected out from the outputs of the body of an existing assertion due to reasoning for class instances with respect to domain or range of a property like in Example 1, where the modified mapping implied assertion is the following:

$$movies(t, d) \rightarrow Director(d)$$

The exact way that this implied assertion will be used depends on the rewriting method, but in any case, in the resulted SQL query column *title* will not be in the *Select* clause.

Situations like this can be identified offline, by analyzing the ontology and the original mapping. The first step is to find the ontology axioms of the form $\exists P.\top \sqsubseteq C$ and $\exists P^-.\top \sqsubseteq C$ (or equivalently $\top \sqsubseteq \forall P.C$) that define the domain and range of some property P to be a class C in our ontology. Then we identify the mapping assertion in $\mathcal{M}$ that generates RDF triples which have as predicate the property P and we modify the target SQL query of the mapping, by projecting out the columns used for the subject or object respectively. At this point, we can skip certain mappings that are covered by the corresponding *rdf:type* mapping for a given class, as OBDA systems eliminate the usage of the property triple pattern for these cases based on foreign key relationships [19]. Consider again the case of Example 1: if we had one more database table $director\_info$ which has a primary key id and we also know that $director$ in $movies$ is a foreign key that references this primary key, then if we also had the mapping:

$$director\_info(id, ...) \rightarrow Director(f(id))$$

the previously obtained mapping can be skipped as it is redundant and will not be used by the OBDA system.

The method is described in Algorithm 1. $ComputeDTR$ is a function that returns the DTR for the query passed as argument. If $DTR = 1$ according to proposition 1, then access to data is avoided altogether, otherwise the actual DTR is computed by sending two count queries: with and without the distinct modifier. Later, when the DTR needs to be determined during query optimization, an estimation based on data summarization is used instead (Section 5). Function $ExistsFK$ returns true if a foreign key exists between the output column of query $query_1$ and the output column of query $query_2$. The result of this algorithm is a set of mapping assertions, possibly annotated with information about the projection of a column. Modification of the produced SQL query in order to take into consideration the views created for these mappings, instead of the original body of the mapping, can simply be performed in the final step of the query unfolding, where each Aux_j is replaced by the corresponding SQL, and as a result it is independent of the query rewriting method.

4 Pushing Duplicate Elimination Before IRI Construction

In SPARQL to SQL approaches, pushing joins inside unions is a well known *structural* optimization, so that joins over IRIs are avoided and relational columns, whose values are possibly indexed, are used instead. Methods for unfolding based in partial datalog evaluation (see Section 2.5) produce such queries, where additionally, union subqueries that contain joins between incompatible IRIs, that when evaluated will produce an empty result, are completely discarded. Also, the *safe separator*[3] of the R2RML mapping language can be used to ensure that concatenation of multiple columns cannot produce the same value with that of a single column [20]. In a similar manner, it can be very useful to perform duplicate elimination before IRI construction. In this section we discuss the process of transforming the unfolded query that has the form shown in formula (1) from Section 2.5, into an equivalent one, such that duplicate elimination is performed on database values.

To do so, we must group together union subqueries that have the same select clause up to variable (column name) renaming. In case of groups that contain only one subquery, Proposition 1 can be used to avoid duplicate elimination altogether. In our case the situation is more complicated, as we want to ensure that tuples produced from different IRI templates cannot possibly have equal values. Consider for example the following query:

[3] https://www.w3.org/TR/r2rml/#dfn-safe-separator

Algorithm 1: Track down SQL queries that contain duplicates

MappingsWithDuplicates (mapping $\mathcal{M}$, ontology $\mathcal{O}$, database schema S, database instance D);
Output: Mapping assertions possibly annotated with a projected column
Uses : $ComputeDTR(query, db\_schema, db\_instance)$
Uses : $ExistsFK(schema, query_1, query_2)$
$result := \emptyset$;
for $m \in \mathcal{M}$ **do**
  if $head(m)$ *is Class assertion* **then**
    if $ComputeDTR(body(m), S, D) > 1$ **then**
      add m to result;
    end
  else
    /* $head(m)$ is Property assertion with predicate P */
    if $\mathcal{O}$ *contains* $\exists P.\top \sqsubseteq C$ **and** $ComputeDTR(\prod_1(body(m)), S, D) > 1$ **and not** ($\exists m2 \in \mathcal{M}$ *s.t. predicate of* $head(m2)$ *is* C **and** $ExistsFK(S, \prod_1(body(m)), body(m2))$ **then**
      add m to *result* for projection of 1st column;
    end
    if $\mathcal{O}$ *contains* $\exists P^{-}.\top \sqsubseteq C$ **and** $ComputeDTR(\prod_2(body(m)), S, D) > 1$ **and not** ($\exists m2 \in \mathcal{M}$ *s.t. predicate of* $head(m2)$ *is* C **and** $ExistsFK(S, \prod_2(body(m)), body(m2))$ **then**
      add m to *result* for projection of 2nd column;
    end
  end
end
return *result*;

```
SELECT ':Person' || alias1.id AS x
 FROM table1 alias1
UNION
SELECT ':Person' || alias1.key AS x
 FROM table2 alias1
UNION
SELECT ':Person' || alias1.id ||
     '/' || alias1.name AS x
 FROM table3 alias1
```

Given that the '/' character is a safe separator, the third subquery cannot produce any result tuple that will be the same with a result tuple coming from the first two subqueries. On the other hand, there is a possibility that the first two subqueries may produce the same answer. The following rewriting of this query can be used:

```
SELECT ':Person' || var1 AS x,
 FROM
 (SELECT
     alias1.id as var1
     FROM
     table1 alias1
   UNION
   SELECT
     alias1.key AS var1
     FROM
     table2 alias1
   )
UNION ALL
SELECT DISTINCT ':Person' ||
     alias1.id || '/' || alias1.name AS x,
 FROM table3 alias1
```

Note that UNION ALL operator is simply concatenating the results. When the UNION ALL is the outer operator of a query, it is reasonable for the RDBMS to start sending the results in a pipelining fashion, as they are produced from each subquery without saving or waiting for all the results to be produced. In this sense, it can be considered a "cheap" operator in contrast to UNION. If we know that column *id* of *table*3 is a primary key, then the distinct keyword of the last subquery can be eliminated. Even if this is not the case, the resulted query has several advantages over the initial. First, the duplicate elimination process has been separated over two distinct result sets and also each tuple is smaller in size. This gives to the RDBMS the opportunity to better utilize available memory, as it now has smaller datasets to perform duplicate elimination, or even parallelize the process. Available indexes on the columns can be used. Also, as discussed, when there is no blocking outer operator, results are produced in a pipelined fashion. This way the first results can be obtained very quickly and, as IRI construction is an expensive operation, the difference can be impressive when we have large results and the processing for each subquery it relatively cheap.

Following the described process, starting from a query as in formula 1, we obtain a query that has the form

$$Query(\boldsymbol{x}) \leftarrow UCQ_1(\boldsymbol{x}) \vee ... \vee UCQ_l(\boldsymbol{x}) \tag{2}$$

where each UCQ_i for $i = 1...l$ is an expression of the form

$$UCQ_i(\boldsymbol{f_i}(\boldsymbol{v_i})) \leftarrow Q_i^1(\boldsymbol{v_i}) \vee ... \vee Q_i^l(\boldsymbol{v_i}) \tag{3}$$

where $\boldsymbol{v_i}$ is a vector of variables not occurring in (1) and $Q_i^1(\boldsymbol{v_i}) \vee ... \vee Q_i^l(\boldsymbol{v_i})$ have been resulted from exactly those conjuncts of (1) that have $\boldsymbol{f_i}$ in the left hand side, by replacing each variable in $\boldsymbol{x_i}$ with the variable in the same position in $\boldsymbol{v_i}$ and adding a conjunction of variable equalities EQ between variables of $\boldsymbol{v_i}$ that correspond to positions of $\boldsymbol{x_i}$ where the same variable occurred. That is, each Q_i^j for $j = 1...k$ has the form:

$$Q_i^j(\boldsymbol{v_i}) \leftarrow Aux_1(\boldsymbol{v_i^1}) \wedge ... \wedge Aux_n(\boldsymbol{v_i^k}) \wedge EQ. \quad (4)$$

In the corresponding SQL query, disjunctions in (2) can be translated to UNION ALL and only disjunctions in (3) need to be translated to UNION (or add DISTINCT if there is only one disjunct) avoiding duplicate elimination over IRIs. Also, EQ can be replaced by choosing one variable for each equality and replacing all occurrences with the specific variable, and then add a renaming operator on the *Select* clause corresponding to $Q_i^j(\boldsymbol{v_i})$.

A problem that may arise is that it is not always possible to perform this structural optimization, because queries with different select clauses can still produce the same result. Suppose that the second subquery in the previous example had the following select clause:

`SELECT ':Person1'||alias1.id AS x.`

As `:Person` is a substring of `:Person1`, the second subquery may still produce the same result as the first one, but in this case we cannot just perform the duplicate elimination on the *id* columns of the two tables. Such cases are peculiar in the sense that a substring relation must hold between constant arguments of different IRI templates. In order to identify exactly when two IRI templates can produce the same result, one should construct a regular expression from each template and then examine if the intersection of the languages produced by these regular expressions is empty. Given that for most cases it is straightforward to make sure that two different IRI templates cannot generate the same result, we instead use a set of quick checks, such that if any of these checks succeed, then it is guaranteed that the IRI templates cannot generate the same result. These include a check for inequality in the number of safe separator characters, a check that the first or last arguments of two templates are not substrings etc. If none of these checks succeed, then we cannot be sure if these IRI templates can produce the same result or not, and we do not push duplicate elimination before IRI construction for the specific subqueries only.

5 Incorporating Decisions for Duplicate Elimination in Query Execution

In this section we consider a query that has the form shown in formula (2) and describe a process that decides about early duplicate elimination. We start by considering separately each union subquery that has the form shown in formula (4). If we consider $Aux_1, ..., Aux_n$ to be relation names belonging to the database schema, this is a CQ, but in practice $Aux_1(\boldsymbol{v_1^i}), ..., Aux_n(\boldsymbol{v_k^1})$ are arbitrary FOL queries. Our method relies on an estimation about the final result size of each union subquery. To obtain this estimation, DTR for every mapping assertion in

$\mathcal{M}$ is no longer enough, as it was the case when creating materialized views, but 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 query of some mapping assertion. As making an estimation for an arbitrary FOL query is an involved process, we make a distinction between assertions in $\mathcal{M}_{CQ}$ and assertions in $\mathcal{M} \setminus \mathcal{M}_{CQ}$. 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.

As a result, we can consider that each union subquery has the form

$$Q(\boldsymbol{x}) \leftarrow Aux_1(\boldsymbol{x_1}) \wedge ... \wedge Aux_n(\boldsymbol{x_j}) \wedge T_1(\boldsymbol{x_{j+1}}) \wedge ... \wedge T_m(\boldsymbol{x_{j+m}}) \tag{5}$$

where each $Aux_1, ..., Aux_j$ corresponds to *body*(m) for some mapping assertion $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$ and $T_1, ..., T_m$ are relation names from the database schema. We will refer to each conjunct in the right hand side of (5) as an *input table* of query $Q(\boldsymbol{x})$.

Let Q be a query as in (5) and $C_i(\boldsymbol{x_i})$ be an input table of Q. The query $ans(\boldsymbol{x_{c_i}}) \leftarrow C_i(\boldsymbol{x_i})$, where $\boldsymbol{x_{c_i}}$ contains exactly the variables of $\boldsymbol{x_i}$ that appear more than once in Q, will be called the *projection query* of input table $C_i(\boldsymbol{x_i})$ from Q. Additionally, let D be a database instance (which will be implied). The DTR of the result of the projection query of $C_i(\boldsymbol{x_i})$ from Q on database instance D will be denoted $DTR^Q_{C_i}$.

Analyzing External Tables. As we operate outside the RDBMS engine, in order to extract the needed information we should import all the corresponding data, something that is clearly not practical. Luckily we have several other options. One first 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. One second 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 depended 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 [8], 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 an offline process, that needs to be done before query execution, similar to an analyze command in a database schema, as it only depends on mappings and data. Adopting the commonly used value independence assumption between the result attributes and the uniformity of values in an attribute [23], 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 estimate that all tuples are distinct.

5.1 Early Duplicate Elimination of Intermediate Results

Let us suppose that we have a single SQL subquery coming from the translation of a SPARQL query 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 5.2. In this case, it may be profitable 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. Uncertainty about query execution costs is an inherent problem in data integration and as the OBDA system operates outside the database engine, knowing all these factors is difficult or even impossible. 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 and will help us take decisions regarding duplicate elimination states that the impact of an input table with DTR equal to a constant number a in the number of tuples of the final query result is proportional to a. As a result of this assumption, the selectivity of the query plays the most important role regarding the 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 if we 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, when we have very selective queries with few results, whereas the size of the intermediate result under consideration is much larger, 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 that has the form (5) and whose result over D is the relation instance R and an input table $C_i(\boldsymbol{x_i})$ of q, then perform duplicate elimination on input table $C_i(\boldsymbol{x_i})$ prior to execution of q if

$$Size_R - \frac{Size_R}{DTR^q_{C_i}} > \frac{Size_A}{DTR^q_{C_i}}$$

where relation instance A is the result of the projection query of $C_i(\boldsymbol{x_i})$ from q on D and $Size_R$ and $Size_A$ are the estimated sizes (in bytes) of relation instances R and A 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.

5.2 Considering Multiple Input Tables and Union Subqueries

When more than one input tables with $DTR > 1$ exist for a query, the decision whether duplicate elimination should be performed for a specific one clearly depends on the decisions that have been taken for the rest, as the size of the result changes depending on the other decisions. In order to avoid examining all the combinations, a greedy approach is used, choosing at first the one that gives the biggest gain according to the heuristic (considering the heuristic as a fraction instead of an inequality). Then the chosen input table is excluded and a new estimation for the final result is made, considering that duplicate elimination is performed on the chosen input table and this step is repeated until none of the rest input tables gives gain according to the heuristic.

When dealing with multiple subqueries of a UNION or UNION ALL query the heuristic must be modified as follows:

$$\sum_{i=1}^{n}(Size_{R_i} - \frac{Size_{R_i}}{DTR_A}) > \frac{Size_A}{DTR_A}$$

where each R_i is the result of a subquery q_i that uses the input table. This modified form should only be used when the the corresponding RDBMS reuses an intermediate result, for example when a temporary table is created. If the result is defined as a nested select subquery instead, then the original form of the heuristic should be used for each subquery separately.

Note that when dealing with multiple subqueries, one has to identify when the projection queries of input tables $C_i(\boldsymbol{x_i})$ coming from different subqueries are equivalent. This process is straightforward, as we are dealing with queries referencing a single input table, and as a result we can construct a single expression consisting of the relation name (for $T_1, ..., T_m$ in (5)) or a mapping assertion ID (for $Aux_1, ..., Aux_n$), the projected columns in ascending order and also in ascending order positions of constants in $\boldsymbol{x_i}$ and the corresponding values and then perform a simple syntactic check for equality.

6 Duplicates Due To Redundant Self-Joins

In this section we identify a case where redundant self-joins lead to introduction of duplicates. These self joins are redundant under standard set semantics, and could be identified by well known conjunctive query minimization methods, without taking into consideration dependencies, such as tableau minimization-known to be NP-hard ([1], Chapter 6.2). We will see that we can identify these

$$ans(f(t), g(d_2), h(a)) \leftarrow \\ movies(t, d_1), movies(t, d_2), movies(t, d_3), hasActor(t, a)$$

(a) Unfolding 1

$$ans(f(t), g(d_2), h(a)) \leftarrow movies(t, d_2), hasActor(t, a)$$

(b) Unfolding 2

Fig. 2: Unfoldings

specific cases without having to resort to general and expensive query minimization. In Section 7.1, we experimentally show how common these cases are.

Self joins of database tables occur frequently in queries produced in SPARQL to SQL translation, as they correspond to the reconstruction of the ternary RDF statements from their reification in terms of relational tuples. Many OBDA systems eliminate this behavior when possible using semantic knowledge in the form of functional dependencies that hold in the database [22, 20]. In case no functional dependency holds for a specific database table, often an indication of a non-normalized schema, the result is the introduction of duplicate answers in the produced SQL query, as it is shown in the next example.

Example 2. Consider the relational tables and the mappings from Figure 1 and the SPARQL query from Figure 1d. Also consider, as in the case of Example 1, that the ontology defines the range of property *hasDirector* to be the class *Director*. This information will be used during rewriting, so that the final unfolded query has the form shown in Figure 2a. In this translation there are three instances of the base relation movies, each one corresponding to the translation of a SPARQL triple pattern. This query evaluated over the given database instance will return 27 answer tuples, whereas the distinct answer tuples are only 9. If the column *title* was unique in table *movies*, then these three instances would be replaced by a single one by an OBDA system that performs the mentioned optimization with respect to functional dependencies, but if it is not, or at least the database metadata do not provide this information, then the self joins would not be avoided. The fact that self joins are needed is already an indication of redundancy, as it implies that a non unique column (or non unique combination of columns) is used to define a class, unless the self join is already present in the SPARQL query. One can observe that the atom $movies(t, d_1)$ that corresponds to the translation of the *rdf:type* triple pattern of the SPARQL query is only contributing to duplicates in the final query and it can be eliminated. The same holds for the atom $movies(t, d_3)$ coming from the translation of the third triple pattern. The query after the elimination is shown in Figure 2b.

An interesting observation is that under SQL bag semantics these queries produce answers with different DTR, whereas under standard CQ containment with set semantics, they are equivalent. An OBDA system that employs semantic optimization techniques for the produced query, such as query minimization using CQ containment would produce the optimized query, but to the best of our knowledge, due to the fact that such a procedure is expensive, none of the current OBDA systems choose to use that. Note that we stress that query minimization should be used for the final query, as the problem shows up after query unfolding with respect to the database mappings, as it is the case for the optimizations that use functional dependencies, so optimization techniques used during query rewriting with respect to the ontological axioms cannot eliminate this problem.

By taking advantage of the fact that we are trying to eliminate only self joins, we can avoid the usage of a general, possibly expensive, query containment algorithm and we only need to examine specific pairs of atoms that have the same predicate, in order to find out if one of them is "covered" by the other.

Let CQ_1 be the conjunctive query $Query(\boldsymbol{x}) \leftarrow R_1(\boldsymbol{x_1}), ..., R_n(\boldsymbol{x_n})$. Atom $R_i(x_i^1, ..., x_i^n)$ is *covered by* atom $R_j(x_j^1, ..., x_j^n)$ in CQ_1 if the following three conditions hold:

- R_i and R_j have the same predicate
- for each x_i^k such that $x_i^k \in Const$, then $x_i^k = x_j^k$
- for each x_i^k such that $x_i^k \in Var$ and x_i^k appears anywhere in $\boldsymbol{x}, \boldsymbol{x_1}, ..., \boldsymbol{x_{i-1}}, \boldsymbol{x_{i+1}}, ..., \boldsymbol{x_n}$, then $x_i^k = x_j^k$

If R_i is covered by R_j then obviously CQ_1 is equivalent to $Query(\boldsymbol{x}) \leftarrow R_1(\boldsymbol{x_1}), ..., R_{i-1}(\boldsymbol{x_{i-1}}), R_{i+1}(\boldsymbol{x_{i+1}}), ..., R_n(\boldsymbol{x_n})$.

In the previous example, both atoms $movies(t, d_1)$ and $movies(t, d_3)$ are covered by $movies(t, d_2)$.

7 Implementation and Experimental Evaluation

We have implemented all the described optimizations in an prototype extension of Ontop version 1.18.0. Our extension is available in github[4], as a fork of the official Ontop repository. All proposed optimizations were added as extra processing phases to the result of the partial datalog evaluation, before the generation of the SQL query. As Ontop uses the so called $\mathcal{T}$-Mappings [19] in order to emulate H-complete ABoxes and perform the tree-witness query rewriting [11] on such ABoxes, we directly use these $\mathcal{T}$-Mappings as our mappings $\mathcal{M}$. In summary, $\mathcal{T}$-Mappings are created from the initial mapping, by compiling in them information about ontological axioms such as property and class hierarchies.

Experimental Setup. All experiments were carried out on a machine with an Intel Core i7-3770K processor with 8 cores and 16 GB of RAM running UBUNTU

[4] https://github.com/dbilid/ontop

14.04 with 3.13.0-95-generic kernel. As our intention was to examine how our optimizations perform in different underlying systems, we used three different back-ends: PostgreSQL (version 9.3), MySQL (version 5.5.52) and one of the most widely used proprietary RDBMS, which due to its license we will call *System I*. All systems were setup and tuned for usage in a machine with 16GB RAM. The schema and data in all systems were identical and all the proposed indexes were created.

7.1 Experiments with NPD and LUBM Benchmarks

We have performed an experimental evaluation of our techniques using the LUBM [7] and NPD [13] benchmarks, with the ontology and mappings that are publicly available at the Ontop repository on github[5] and with existential reasoning enabled. Both datasets were generated for scale 100. In PostgreSQL database size was about 1.1 GB for LUBM and about 5.8 GB for NPD.

Mappings. For LUBM benchmark in total **84** mapping assertions were produced as $\mathcal{T}$-Mappings from Ontop. Out of these, we only needed to make DTR estimations using statistics for **15**, as we found all the others to have DTR=1 according to Proposition 1. Offline gathering and building of statistics for the columns referenced in the candidate mapping assertions took **28** sec. Making DTR estimations for the result of the 15 mappings took less that **0.5** seconds. For NPD, out of **1091** produced assertions in $\mathcal{T}$-Mappings **112** where found to have $DTR > 1$. Building statistics took **45** sec. and making DTR estimations about **2** sec.

Queries. For LUBM we used the original 14 queries. Covered self-join elimination was applicable to **6** queries, namely: 2, 4, 7, 8, 9 and 12. Early duplicate elimination was applicable to **4** queries: 2, 4, 7 and 9. Out of these, 4 and 7 were not finally found beneficial from our heuristic. For NPD we used a subset of 21 out of the original 30 queries: queries 1-14, 22-25 and 28-30, excluding the queries that use GROUP BY, as it is not supported by the used Ontop version, and queries that produce empty SQL due to incompatible IRIs. To these queries we added four more, in order to showcase the advantage of duplicate elimination. 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. Out of the final 25 NPD queries, covered self-join elimination was applicable to **10** queries: 6, 9, 10, 11, 12, 23, 24, 29, 30 and 32. Early duplicate elimination was applicable to **6** queries: 9, 24, 31, 32, 33 and 34. Duplicate elimination for query 9

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

was not performed according to our heuristic. For all the queries, pushing duplicate elimination before IRI construction was performed, according to Section 4. All SPARQL queries were executed using the DISTINCT modifier.

Overhead in Optimization. Total optimization time for the 14 LUBM queries increased from **533** ms to **948** ms, whereas for the 25 NPD queries the increase was from **765** ms to **1163** 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, whereas the second time is the time needed by our modified version that performs all the described optimizations.

Results. We executed the queries with and without the proposed optimizations. For each query we used a timeout of 20 minutes. 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. For early duplicate elimination, temporary tables were used during execution in the same session as the main query and unique indexes were created on those tables. All times are in milliseconds. For each query we present the time needed to obtain the first 1000 (that was the JDBC ResultSet fetch size used) answers (First) and all the answers (Total) without the optimizations and with the optimizations (First Opt and Total Opt). All results and the produced SQL queries are available in our repository in github[6].

Results are presented in Table 1 for PostgreSQL, Table 2 from System I and Table 3 for MySQL. For PostgreSQL without the optimizations we had three timeouts with average time (excluding the timeouts) for first answers **27154** ms and for all answers **27612**. With the optimizations no timeout occurred and the times were **8337** ms for first answers and **16561** for all answers. System I was the only system that did not have any timeouts. Without the optimizations we had average time for first answers **36761** and for all answers **37788** ms. With the Optimizations the times were **8612** and **14222** ms. In MySQL without the optimizations we had four timeouts with average time (excluding the timeouts) for first answers **36982** ms and for all answers **37210**. With the Optimizations we had two timeouts and the times (excluding the timeouts) were **23621** and **23682** ms. MySQL is the only system that does not take advantage of result pipelining. Furthermore, even in the optimized setting we had two timeouts for NPD queries 32 and 34. On closer inspection, some composite indexes were found to be used inefficiently. We could solve this problem by creating four new non-composite indexes, but all the times given, including those in Section 7.2, are without this modified design.

[6] https://github.com/dbilid/ontop/tree/version3/results

	First	Total	First Opt	Total Opt
L1	326	326	390	390
L2	timeout	timeout	3340	3340
L3	52	52	61	61
L4	1619	1619	519	519
L5	35	36	12	12
L6	20833	21508	1242	8389
L7	1130	1130	257	257
L8	1111	1118	204	377
L9	144595	144639	6953	61717
L10	109	109	24	24
L11	45	45	35	35
L12	9	9	11	11
L13	177	177	191	191
L14	4348	4822	15	4210
N1	5464	6931	1740	6693
N2	4481	4646	4546	4668
N3	1067	1121	1020	1095
N4	31428	32689	30726	31999
N5	98	98	87	87
N6	66267	67360	36614	37712
N7	1086	1086	1086	1087
N8	399	409	416	429
N9	3302	3311	2892	2899
N10	5539	5579	5577	5621
N11	48675	49039	28412	28754
N12	90857	91548	65251	65954
N13	1603	1806	1015	1738
N14	33872	36984	21299	35646
N22	8272	8907	346	4527
N23	10521	11053	355	4699
N24	5129	5211	227	547
N25	13879	14842	21	13707
N28	39465	41362	8458	31973
N29	86875	87351	52563	56161
N30	162341	163322	31265	105459
N31	5470	6595	1657	4380
N32	timeout	timeout	4694	80775
N33	177073	177192	351	371
N34	timeout	timeout	11281	39365
Avg.	27154[1]	27612[1]	8337	16561

[1] excluding timeouts

Table 1: PostgreSQL Results (ms)

	First	Total	First Opt	Total Opt
L1	424	424	396	397
L2	4239	4240	5966	5967
L3	96	97	84	84
L4	17894	17894	3253	3253
L5	41	43	29	30
L6	12317	13842	1096	5833
L7	1525	1526	505	505
L8	620	633	148	211
L9	69961	70061	11830	42709
L10	11	11	8	8
L11	76	76	83	84
L12	17	17	17	17
L13	367	367	236	236
L14	2246	3063	15	2510
N1	5394	8193	8655	11468
N2	796	1008	1915	2171
N3	162	239	294	411
N4	1491	3697	3051	5544
N5	79	79	110	110
N6	546642	548704	24133	26511
N7	1160	1160	2678	2679
N8	175	179	215	222
N9	2800	2810	3355	3371
N10	5257	5339	4898	4955
N11	33257	34030	32246	33010
N12	60507	62182	41033	42536
N13	2930	3281	1224	3751
N14	52227	57246	4253	47430
N22	6766	8345	356	5069
N23	10217	11484	814	4499
N24	7188	7544	1473	1772
N25	6558	8473	16	6626
N28	34468	38980	8234	29598
N29	114412	116586	50755	54148
N30	196992	200680	85099	114896
N31	9177	10214	5777	8046
N32	130472	131039	5492	34277
N33	11598	11619	774	792
N34	83118	88344	4836	28408
Avg.	36761	37788	8086	13040

Table 2: System I Results (ms)

	First	Total	First Opt	Total Opt
L1	431	432	407	408
L2	441148	441149	8458	8459
L3	235	236	264	264
L4	6537	6537	1718	1718
L5	16	17	27	28
L6	27664	27915	25504	25571
L7	3268	3268	264	264
L8	17488	17490	761	761
L9	timeout	timeout	288259	288262
L10	6	7	4	4
L11	69	70	7	7
L12	6	6	5	5
L13	135	136	136	137
L14	5570	5734	2309	2363
N1	70806	71477	69467	69581
N2	2532	2766	2605	2617
N3	900	969	1132	1137
N4	23658	24996	18315	18415
N5	120	121	128	128
N6	212227	213339	53208	54381
N7	30864	30864	30366	30366
N8	578	580	605	605
N9	10880	10890	9065	9066
N10	2925	2979	2042	2046
N11	14187	14555	18155	18181
N12	25576	26297	37554	37606
N13	20573	20648	24624	24640
N14	39836	40001	37651	37816
N22	13218	13476	7427	7483
N23	15814	16067	6416	6460
N24	9964	10011	1331	1345
N25	19878	20285	10064	10161
N28	48142	49133	45359	45493
N29	81239	81459	58781	58817
N30	140646	141015	103019	103074
N31	7232	7423	2671	2707
N32	timeout	timeout	timeout	timeout
N33	timeout	timeout	5873	5876
N34	timeout	timeout	timeout	timeout
Avg.	36982[1]	37210[1]	23621[1]	23682[1]

[1] excluding timeouts

Table 3: MySQL Results (ms)

Considering Duplicate Elimination Only. In order to isolate the impact of early duplicate elimination, we have isolated the queries such that this optimization is applicable, and we have executed them with the optimizations for pushing IRI construction before duplicate elimination and covered atom elimination disabled. Note that since other optimizations are not applied, for queries NPD 9 and NPD 24 the decision made by our algorithm has changed, that is, with all the optimizations enabled DE was chosen to be performed, whereas now it was not, so they are not included in the results. Results for this setting for all systems are presented in Table 4. As before all the produced queries are available on our github repository. The results for PostgreSQL and MySQL confirm our assumption that relational engines do not consider such optimization options. For System I, some results, like for example for query LUBM 2, are more intriguing. On a closer inspection, we saw that indeed in some cases SystemI does take advantage of early duplicate elimination opportunities. By examining the execution plans we deduced that it does so when an index can be used for all projected columns and this can be done "effortless" during preparation for a subsequent join, for example during hashing in the build phase of a hash join.

In any case, from our experiments it seems that even System I misses many opportunities for early DE that lead to better plans and certainly does not consider using the same duplicate elimination result for different union subqueries. For all queries that our algorithm chose duplicate elimination, the total execution time (without the other optimizations) for System I is **179945** milliseconds, whereas without duplicate elimination it's **315417** milliseconds. Learning exactly when System I considers early DE can lead to even larger improvement, if taken into consideration from our method as system-specific optimization.

	SysI	SysI-DE	PostgreSQL	PostgreSQL-DE	MySQL	MySQL-DE
L2	4240	5260	timeout	3584	441149	7440
L9	69961	52893	144639	77174	timeout	396194
N31	10214	8948	6595	5967	7423	5625
N32	131039	53710	timeout	131701	timeout	timeout
N33	11619	910	177192	1643	timeout	7970
N34	88344	58224	timeout	84710	timeout	timeout
Total Time	315417	179945	>3928426	304779	>5248572	>2817229

Table 4: Execution Times (ms) With Duplicate Elimination Only

7.2 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. 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 were 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. Exact times and queries can be found at our github repository.

8 Related Work and Conclusions

In this work, we considered query reformulations such that the final SQL query is a union of CQs. In [3] a cost based comparison of different reformulations is carried out, considering that the database stores the data as triples, that is no unfolding with respect to mapping assertions is carried out. In general, the final SQL query will be a join of unions of CQs. The authors also note the problem of large intermediate results due to duplicates, and they always perform duplicate elimination on the result of each union of CQs. We believe that our work can be combined with the aforementioned method, in order to

	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
query-mix	PostgreSQL	167116	618328	144984	146406	143191
	MySQL	1129311	-	1066499	1056659	1056145
	SystemI	135790	520408	102724	101984	99989

Table 5: Query Results for Different Duplicate Elimination Strategies

push duplicate elimination to certain input tables only and avoid unnecessary operations. Recently, an extension of this work for arbitrary relational schemas has been presented [14].

In [9] the authors 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. We believe this is orthogonal to our approach as presented in Section 4, as disjointness axioms can be used to further split query groups.

Ultrawrap-OBDA is presented in [21]. From the description of the system, it seems that the saturated mappings used will also suffer from the problem of duplicates in the case of the inference rules (A, dom, B) and (A, range, B). After the identification of these mappings, if Ultrawrap decides to materialize them, duplicate elimination should be used, or in the case that these mappings are not materialized, we believe that our heuristic is straightforward to be applied.

[6] presents 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's conceptually closer to the relational model and does not need separate mappings, so a separate unfolding phase is not needed.

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

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc., 1995.
2. Dina Bitton and David J DeWitt. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)*, 8(2):255–265, 1983.
3. Damian Bursztyn, François Goasdoué, and Ioana Manolescu. Teaching an RDBMS about ontological constraints. *Proceedings of the VLDB Endowment*, 9(12):1161–1172, 2016.
4. Surajit Chaudhuri and Moshe Y Vardi. Optimization of real conjunctive queries. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 59–70. ACM, 1993.
5. Alexandros Chortaras, Despoina Trivela, and Giorgos B Stamou. Optimized query rewriting for owl 2 ql. In *CADE*, volume 11, pages 192–206. Springer, 2011.
6. Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Query rewriting and optimization for ontological databases. *ACM Transactions on Database Systems (TODS)*, 39(3):25, 2014.
7. Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.
8. Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 19–30. VLDB Endowment, 2003.
9. Jason St Jacques, David Toman, and Grant E Weddell. Object-relational queries over $CFD_{nc}^{\forall-}$ knowledge bases: OBDA for the SQL-literate. In *Description Logics*, 2016.
10. Evgeny Kharlamov, Dag Hovland, Ernesto Jiménez-Ruiz, Davide Lanti, Hallstein Lie, Christoph Pinkel, Martin Rezk, Martin G Skjæveland, Evgenij Thorstensen, Guohui Xiao, et al. Ontology based access to exploration data at Statoil. In *International Semantic Web Conference*, pages 93–112. Springer, 2015.
11. Stanislav Kikot, Roman Kontchakov, and Michael Zakharyaschev. Conjunctive query answering with owl 2 ql. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2012.
12. Henning Koehler and Sebastian Link. Armstrong axioms and boyce–codd–heath normal form under bag semantics. *Information processing letters*, 110(16):717–724, 2010.
13. Davide Lanti, Martin Rezk, Guohui Xiao, and Diego Calvanese. The NPD benchmark: Reality check for OBDA systems. In *Proc. of the 18th Int. Conf. on Extending Database Technology (EDBT)*, 2015.
14. Davide Lanti, Guohui Xiao, and Diego Calvanese. Cost-driven ontology-based data access. 2017.
15. Glenn N Paulley and Per-Åke Larson. Exploiting uniqueness in query optimization. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: distributed computing-Volume 2*, pages 804–822. IBM Press, 1993.
16. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.
17. Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. Efficient query answering for owl 2. *The Semantic Web-ISWC 2009*, pages 489–504, 2009.

18. Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on data semantics*, pages 133–173. Springer, 2008.
19. Mariano Rodriguez-Muro, Roman Kontchakov, and Michael Zakharyaschev. Ontology-based data access: Ontop of databases. In *International Semantic Web Conference*, pages 558–573. Springer, 2013.
20. Mariano Rodríguez-Muro and Martin Rezk. Efficient SPARQL-to-SQL with R2RML mappings. *Web Semantics: Science, Services and Agents on the World Wide Web*, 33:141–169, 2015.
21. Juan F Sequeda, Marcelo Arenas, and Daniel P Miranker. OBDA: query rewriting or materialization? in practice, both! In *International Semantic Web Conference*, pages 535–551. Springer, 2014.
22. Juan F Sequeda and Daniel P Miranker. Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 22:19–39, 2013.
23. Arun Swami and K Bernhard Schiefer. On the estimation of join result sizes. In *International Conference on Extending Database Technology*, pages 287–300. Springer, 1994.

A NPD Queries 31-34

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

Listing 1.1: Query NPD 31

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

Listing 1.2: Query NPD 32

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

Listing 1.3: Query NPD 33

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

Listing 1.4: Query NPD 34