Efficient Duplicate Elimination in SPARQL to SQL Translation

Dimitris Bilidas and Manolis Koubarakis

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

Abstract. Redundant data 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 an approach for data integration 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 table from Figure 1a and the mapping from Figure 1c. In this mapping *hasDirector* is a property defined in the ontology, whereas f and g are functions 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 a basic graph pattern on this graph can never result in duplicate answers [7]. 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 triple patterns, 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* (DE) 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 DE is a costly blocking operation [1] 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 DE options during optimization is not usually regarded worthy. Contrary to this situation for SQL queries, it has been ascertained [4] 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.

In a normalized relational schema 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 during evaluation of 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, which can lead to a large number of duplicates as illustrated in Example 1.

Example 1. Consider the mapping from Figure 1c 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

title1	director1
title2	director1
title1	director2

(a) Table movies(title, director)

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

(b) SQL query

$$movies(t, d) \rightarrow hasDirector(f(t), g(d))$$

(c) Example Mapping

Fig. 1: Movies Database

condition that this column should not be null. The resulting SQL to be sent for evaluation is shown in Figure 1b.

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, DE 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) unindexed string values.

In this work we present efficient solutions to the described problems, considering ontologies belonging to the OWL 2 QL language, as the W3C recommendation for query answering against datasets stored in relational back-ends. We start by providing some preliminaries regarding ontologies, mappings and relational databases (Section 2) and 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). Duplicate-free 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 DE 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 DE of intermediate results. Having implemented our optimizations in the state of the art OBDA system Ontop [9], in Section 6 we present their impact on NPD and LUBM benchmarks using four 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. Excluding the queries that give a timeout in the unoptimized setting, the overall improvement for the rest of the queries can be more than 60%. 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%.

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 [8], 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.

Databases and Queries. We start by giving definitions for database instances and queries over them, following the bag semantics from [2]. A *bag* B is a pair (A, μ), where A is a set called the underlying set of B and μ 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 by 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.

A *SQL query* over a relational schema S is an expression of 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 by $\prod_i(q)$ the query resulting from the projection of the i-th answer term of q, that is the query: $Query(x_i) \leftarrow \alpha$. The *duplicate-tuple ratio* DTR_R of a relation instance R is 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.

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 [6]:

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 $X \to Y$ 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.*

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 filling assertions $P(\rho, \rho')$, where $\rho, \rho' \in \mathcal{O}_{Const}$ 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: $\phi(\boldsymbol{x}) \rightarrow \psi$, where $\phi(\boldsymbol{x})$ will be denoted by *body(m)* and it is a SQL query over a database instance D, ψ has the form $P(cc^1(\boldsymbol{x^1}), cc^2(\boldsymbol{x^2}))$ or $C(cc^1(\boldsymbol{x^1}))$ with P (respectively C) $\in \Sigma_O$ a property (respectively concept) name, and each $cc^j \in \Lambda$ is a function with arity equal to the length of $\boldsymbol{x^j}$ and range a subset of $\mathcal{O}_{Const}$. All variables in ψ also appear in $\boldsymbol{x}$. The right hand side will be denoted by *head(m)*. A *mapping* $\mathcal{M}$ is a finite set of such mapping assertions. Note that according to this definition, in this paper we consider only *global-as-view* (GAV) mappings. 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.

Query Rewriting and Unfolding As we mentioned in Section 1, query answering in OBDA involves 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 (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 [8]. 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 this step is a UCQ over the ontology.

Regarding query unfolding with respect to a mapping $\mathcal{M}$, a method based on partial Datalog evaluation with functional terms is presented in [8]. 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 DE 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 database size. Also, this solution will not take into consideration duplicates due to projections in the SPARQL query and finally, it 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. 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 [9]. 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 in order to collect statistical information 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

Algorithm 1: Track down SQL queries that contain duplicates

```
GetDuplicates (mapping $\mathcal{M}$, ontology $\mathcal{O}$, database schema $S$, database instance $D$);
Output: Mapping assertions possibly annotated with a projected column
$result := \emptyset$;
for $m \in \mathcal{M}$ do
    if $head(m)$ is Class assertion then
        if $ComputeDTR(body(m), S, D) > 1$ then add $m$ to $result$;
    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;
        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
return $result$;
```

summarization is used instead (Section 5). Function $ExistsFK$ returns true if a foreign key exists between the output column of the query passed as second argument and the output column of the query passed as third argument. 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 DE 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) produce such queries, where additionally, union subqueries that contain joins between incompatible IRIs, that when evaluated will produce an empty result, are completely discarded. In a similar manner, it can be very useful to perform DE before IRI construction. In this section we discuss the process of transforming the unfolded query that has the form shown in formula (1) at page 5, into an equivalent one, such that DE 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. 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 $Q_i^1(\boldsymbol{v_i}) \vee ... \vee Q_i^l(\boldsymbol{v_i})$ result 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

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

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 DE 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 $Q_i^j(\boldsymbol{v_i})$. 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. Also, the DISTINCT keyword can be entirely avoided in some cases where Proposition 1 is applicable. Even if this is not the case, the resulted query has several advantages over the initial. First, the DE process has been separated over multiple result subsets 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 DE, 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 with first results 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 is relatively cheap.

5 Incorporating Decisions for DE 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 DE. 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 SQL 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 SQL 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. Adopting the commonly used value independence assumption between the result attributes and the uniformity of values in an attribute [11], 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.

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

$$Q(\boldsymbol{x}) \leftarrow Aux_1(\boldsymbol{x_1}) \wedge ... \wedge Aux_n(\boldsymbol{x_n}) \wedge R_1(\boldsymbol{x_{n+1}}) \wedge ... \wedge R_m(\boldsymbol{x_{n+m}}) \quad (5)$$

where each $Aux_1, ..., Aux_n$ corresponds to $body(m)$ for some mapping assertion $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$ and $R_1, ..., R_m$ 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 $I_i(\boldsymbol{x_i})$ be an input table of q. The query $ans(\boldsymbol{x_{c_i}}) \leftarrow I_i(\boldsymbol{x_i})$, 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 $I_i(\boldsymbol{x_i})$ from q.

Let us suppose for now 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. In this case, it may be beneficial to dictate the RDBMS to perform the DE 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 on 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 DE 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 regarding the DE decisions. Intuitively, a query whose result size is much larger than the size of the intermediate result for which we examine the DE 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 DE, especially when expecting the optimizer to limit the sizes of intermediate query results as soon as possible.

A Heuristic Regarding DE. Given a database instance D, a query q that has the form (5) and whose result over D is the relation instance Q and an input table $I_i(\boldsymbol{x_i})$ of q, then perform DE on input table $I_i(\boldsymbol{x_i})$ 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 $I_i(\boldsymbol{x_i})$ from q on D and $Size_Q$ and $Size_{Ans}$ are the estimated sizes of relation instances Q and Ans respectively. That is, DE 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 DE.

When more than one input tables with $DTR > 1$ exist for a query 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 DE 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. Also, when dealing with multiple subqueries of a UNION query the heuristic must be modified such that the left hand side is replaced by a sum over all subqueries. Details can be found in a technical report available online[1].

6 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[2], as a fork of the official Ontop repository. 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 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 RDBM systems, which due to their license we will call *System I* and *System X*. We have performed an experimental evaluation of our techniques using the LUBM [3] and NPD [5] benchmarks. 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

[1] https://github.com/dbilid/ontop/tree/version3/results/de-tech.pdf

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

112 where found to have $DTR > 1$. Building statistics took **45** sec. and making DTR estimations about **2** sec.

Effects of Early DE. We examined the effect of early DE on several queries from these two benchmarks such that our optimization is applicable. In order to isolate the impact of early DE, we executed the queries with the optimization for pushing IRI construction before DE disabled. Results for all systems are presented in Table 1, where SI-D stands for System-I with only early DE enabled and SI for the execution without any optimization in System I. Same holds for System X (SX), PostgreSQL (PG) and MySQL (MS). For all systems, queries for both settings produced identical results. All produced queries are available in github. From the results we can see that the impact of early DE varies depending on each query and each system. For example, in query USER19 (from [10] coming from user requirements), the decision leads to a slight increase in execution time for both systems I and X, but it is very beneficial for PostgreSQL. On a closer inspection, we saw that both these systems do take advantage of early DE opportunities in some cases, such as when distinct rows can be obtained without extra effort using an index of a base table for all projected columns. Learning exactly when these system consider early DE can lead to even larger improvement, if it is taken into consideration from our method as system-specific optimization. In any case, from our experiments it seems that even these systems miss many opportunities for early DE that lead to better plans and certainly do not consider using the same duplicate elimination result for different union subqueries. As a result, even with the system-agnostic version, our method effectively eliminates most timeouts. Even in System X, where no timeout occurs in the unoptimized setting, early DE alone leads to a decrease of **38%** in total execution time.

Query	SI-D	SI	SX-D	SX	PG-D	PG	MS-D	MS
NPD24	7	8.3	7.99	8.09	5.35	5.71	10.8	11.1
NPD31	49.2	T/O	158	174	72.8	T/O	T/O	T/O
NPD32	4.31	8.18	1.46	5.16	0.72	191	8.31	T/O
NPD33	116	170	177	295	108	T/O	T/O	T/O
NPD34	11.2	11.4	7.44	6.92	4.23	9.78	6.23	7.66
USER13	147	157	113	315	T/O	T/O	T/O	T/O
USER19	2.85	2.16	1.88	1.44	3.16	32.3	10.9	144
LUBM2	14.1	12.4	15.4	14.3	4.27	T/O	8.28	488
LUBM9	214	280	65.3	69.6	78.6	153	352	T/O
Total	566	>649	547	890	>277	>392	>397	>651

Table 1: Effect of Early Duplicate Elimination (Times in sec.)

We have also evaluated our DE heuristic by applying extra filters in several query fragments upon which DE is applicable, in order to test our heuristic with different selectivities. We have executed these queries following three different strategies regarding DE: always-perform, never-perform and perform according to heuristic. The never-perform has the worst performance overall due to low selectivity queries dominating the execution times, but it is often better than never-perform for highly selective queries. Our heuristic is always better from the other two strategies on average for all tested queries, leading to an improvement of up to 25% for query mixes where low selectivity queries do not dominate execution time.

References

1. Dina Bitton and David J DeWitt. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)*, 8(2), 1983.
2. Surajit Chaudhuri and Moshe Y Vardi. Optimization of real conjunctive queries. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART*. ACM, 1993.
3. 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), 2005.
4. Evgeny Kharlamov, Dag Hovland, Ernesto Jiménez-Ruiz, Davide Lanti, Hallstein Lie, Christoph Pinkel, Martin Rezk, Martin G Skjæveland, Evgenij Thorstensen, Guohui Xiao, Dmitriy Zheleznyakov, and Ian Horrocks. Ontology based access to exploration data at Statoil. In *International Semantic Web Conference*. Springer, 2015.
5. 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.
6. 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*. IBM Press, 1993.
7. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3), 2009.
8. Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on data semantics*. Springer, 2008.
9. Mariano Rodriguez-Muro, Roman Kontchakov, and Michael Zakharyaschev. Ontology-based data access: Ontop of databases. In *International Semantic Web Conference*. Springer, 2013.
10. Martin G Skjæveland, Espen H Lian, and Ian Horrocks. Publishing the norwegian petroleum directorates factpages as semantic web data. In *International Semantic Web Conference*. Springer, 2013.
11. Arun Swami and K Bernhard Schiefer. On the estimation of join result sizes. In *International Conference on Extending Database Technology*. Springer, 1994.