OBDA Constraints for Effective Query Answering

Dag Hovland[1], Davide Lanti[2], Martin Rezk[2], and Guohui Xiao[2]

[1]University of Oslo, Norway
[2]Free University of Bozen-Bolzano, Italy

Abstract. In Ontology Based Data Access (OBDA) users pose SPARQL queries over an ontology that lies on top of relational datasources. These queries are translated on-the-fly into SQL queries by OBDA systems. Standard SPARQL-to-SQL translation techniques in OBDA often produce SQL queries containing redundant joins and unions, even after a number of semantic and structural optimizations. These redundancies are detrimental to the performance of query answering, especially in complex industrial OBDA scenarios with large enterprise databases. To address this issue, we introduce two novel notions of OBDA constraints and show how to exploit them for efficient query answering. We conduct an extensive set of experiments on large datasets using real world data and queries, showing that these techniques strongly improve the performance of query answering up to orders of magnitude.

1 Introduction

In Ontology Based Data Access (OBDA) [18], the complexity of data storage is hidden by a conceptual layer on top of an existing relational database (DB). Such a conceptual layer, realized by an ontology, provides a convenient vocabulary for user queries, and captures domain knowledge (e.g., hierarchies of concepts) that can be used to enrich query answers over incomplete data. The ontology is connected to the relational database through a declarative specification given in terms of mappings that relate each term in the ontology (each class and property) to a (SQL) view over the database. The mappings and the database define a *(virtual)* RDF graph that, together with the ontology, can be queried using the *SPARQL query language*.

To answer a SPARQL query over the conceptual layer, a typical OBDA system translates it into an equivalent SQL query over the original database. The translation procedure has two major stages: (1) *rewriting* the input SPARQL query with respect to the ontology and (2) *unfolding* the rewritten query with respect to the mappings. A well-known theoretical result is that the size of the translation is worst-case exponential in the size of the input query [13]. These worst-case scenarios are not only theoretical, but they also occur in real-world applications, as shown in [16], where some user SPARQL queries are translated into SQL queries containing thousands of join and union operators. This is mainly due to *(i)* SPARQL queries containing joins of ontological terms with rich hierarchies, which lead to redundant unions [19]; and *(ii)* reifications of n-ary relations in the database into triples over the RDF data model, which lead to SQL translations containing several (mostly redundant) self-joins. How to reduce the impact of exponential blow-ups through optimization techniques so as to make OBDA applicable to real-world scenarios is one of the main open problems in current OBDA research.

The standard solutions to tackle this problem are based on *semantic and structural optimizations* [19,20] originally from the database area [5]. Semantic optimizations use explicit integrity constraints (such as primary and foreign keys) to remove redundant joins and unions from the translated queries. Structural optimizations are in charge of reshaping the translations so as to take advantage of database indexes.

The main problem addressed in this paper is that these optimizations cannot exploit constraints that go beyond database dependencies, such as domain constraints (e.g., people have only one age, except for Chinese people who have two ages), or storage policies in the organization (e.g., table `married` *must* contain all the married employees). We address this problem by proposing two novel classes of constraints that go beyond database dependencies. The first type of constraint, *exact predicate*, intuitively describes classes and properties whose elements can be retrieved without the help of the ontology. The second type of constraint, *virtual functional dependency* (VFD), intuitively describes a functional dependency over the virtual RDF graph exposed by the ontology, the mappings, and the database. These notions are used to enrich the OBDA specification so as to allow the OBDA system to identify and prune redundancies from the translated queries. To help the design of enriched specifications, we provide tools that detect the satisfied constraints within a given OBDA instance. We extend the OBDA system *Ontop* so as to exploit the enriched specification, and evaluate it in both a large-scale industrial setting provided by the petroleum company Statoil, and in an ad-hoc artificial and scalable benchmark with different commercial and free relational database engines as back-ends. Both sets of experiments reveal a drastic reduction on the size of translated queries, which in some cases is reduced by orders of magnitudes. This allows for a major performance improvement of query answering.

The rest of the paper is structured as follows: Preliminaries are provided in Section 2. In Section 3 we describe how state-of-the-art OBDA systems work, and highlight the problems with the current optimization techniques. In Section 4 we formally introduce our novel OBDA constraints, and show how they can be used to optimize translated queries. In Section 5 we provide an evaluation of the impact of the proposed optimization techniques on the performance of query answering. In Section 6 we briefly survey other related works. Section 7 concludes the paper. Omitted proofs and extended experiments with Wisconsin benchmark can be found in the extended version of this paper [12].

2 Preliminaries

We assume the reader to be familiar with relational algebra and SQL queries, as well as with ontology languages and in particular with the OWL 2 QL[1] profile. To simplify the notation we express OWL 2 QL axioms by their description logic counterpart *DL-Lite*$_\mathcal{R}$ [4]. Notation-wise, we will denote tuples with the bold faces; e.g., $\mathbf{x}$ is a tuple.

Ontology and RDF Graphs. The building block of an ontology is a *vocabulary* (N_C, N_R), where N_C, N_R are respectively countably infinite disjoint sets of *class names* and (object or datatype) *property names*. A *predicate* is either a class name or a property name. An *ontology* is a finite set of axioms constructed out a vocabulary, and it describes a domain of interest. These axioms of an ontology can be serialized into a concrete syntax. In the following we use the *Turtle syntax* for readability.

[1] `http://www.w3.org/TR/owl2-overview/`

Example 1. The ontology from Statoil captures the domain knowledge related to oil extraction activities. Relevant axioms for our examples are:

```
:isInWell rdfs:domain :Wellbore                    :isInWell rdfs:range :Well
:hasInterval rdfs:domain :Wellbore                 :hasInterval rdfs:range :WellboreInterval
:completionDate rdfs:domain :Wellbore
:ProdWellbore rdfs:subClassOf :DevelopWellbore     :DevelopWellbore rdfs:subClassOf :Wellbore
```

The first five axioms specify domains and ranges of the properties `:isInWell`, `:hasInterval`, and `:completionDate`. The last two state the hierarchy between different wellbore[2] classes.

Given a countably infinite set N_I of *individual names* disjoint from N_C and N_R, an *assertion* is an expression of the form $A(i)$ or $P(i_1, i_2)$, where $i, i_1, i_2 \in N_I$, $A \in N_C$, $P \in N_R$. An OWL 2 QL *knowledge base* (KB) is a pair $(\mathcal{T}, \mathcal{A})$ where $\mathcal{T}$ is an OWL 2 QL ontology and $\mathcal{A}$ is a set of assertions (also called ABox). Semantics for entailment of assertions ($\models$) in OWL 2 QL KBs is given through Tarski-style interpretations in the usual way [1]. Given a KB $(\mathcal{T}, \mathcal{A})$, the *saturation of $\mathcal{A}$ with respect to $\mathcal{T}$* is the set of assertions $\mathcal{A}_{\mathcal{T}} = \{A(s) \mid (\mathcal{T}, \mathcal{A}) \models A(s)\} \cup \{P(s, o) \mid (\mathcal{T}, \mathcal{A}) \models P(s, o)\}$. In the following, it is convenient to view assertions $A(s)$ and $P(s, o)$ as the *RDF triples* $(s, \texttt{rdf:type}, A)$ and (s, P, o), respectively . Hence, we view a set of assertions also as an RDF graph $\mathcal{G}^{\mathcal{A}}$ defined as $\mathcal{G}^{\mathcal{A}} = \{(s, \texttt{rdf:type}, A) \mid A(s) \in \mathcal{A}\} \cup \{(s, P, o) \mid P(s, o) \in \mathcal{A}\}$. Moreover, the *saturated RDF graph* $\mathcal{G}^{(\mathcal{T},\mathcal{A})}$ associated to a knowledge base $(\mathcal{T}, \mathcal{A})$ consists of the set of triples entailed by $(\mathcal{T}, \mathcal{A})$, i.e. $\mathcal{G}^{(\mathcal{T},\mathcal{A})} = \mathcal{G}^{\mathcal{A}_{\mathcal{T}}}$.

OBDA and Mappings. Given a vocabulary (N_C, N_R) and a database schema Σ, a *mapping* is an expression of the form $A(f_1(\mathbf{x}_1)) \leftarrow sql(\mathbf{y})$ or $P(f_1(\mathbf{x}_1), f_2(\mathbf{x}_2)) \leftarrow sql(\mathbf{y})$, where $A \in N_C$, $P \in N_R$, f_1, f_2 are function symbols, $\mathbf{x}_i \subseteq \mathbf{y}$, for $i = 1, 2$, and $sql(\mathbf{y})$ is an SQL query in Σ having output attributes $\mathbf{y}$. Given Q in $N_C \cup N_R$, *a mapping m is defining Q* if Q is on the left hand side of m.

Given an SQL query q and a DB instance D, q^D denotes the set of answers to q over D. Given a database instance D, and a set of mappings $\mathcal{M}$, we define the *virtual assertions set* $\mathcal{A}_{\mathcal{M},D}$ as follows:

$$\mathcal{A}_{\mathcal{M},D} = \{A(f(\mathbf{o})) \mid \mathbf{o} \in \pi_{\mathbf{x}}(sql(\mathbf{y}))^D \text{ and } A(f(\mathbf{x})) \leftarrow sql(\mathbf{y}) \text{ in } \mathcal{M}\} \quad \cup$$
$$\{P(f(\mathbf{o}), g(\mathbf{o'})) \mid (\mathbf{o}, \mathbf{o'}) \in \pi_{\mathbf{x}_1,\mathbf{x}_2}(sql(\mathbf{y}))^D \text{ and } P(f(\mathbf{x}_1), g(\mathbf{x}_2)) \leftarrow sql(\mathbf{y}) \text{ in } \mathcal{M}\}$$

In the Turtle syntax for mappings, we use *templates*–strings with placeholders–for specifying the functions (like f and g above) that map database values into URIs and literals. For instance, the string `<http://statoil.com/{id}>` is a URI template where "`id`" is an attribute; when `id` is instantiated as "`1`", it generates the URI `<http://statoil.com/1>`.

An *OBDA specification* is a triple $\mathcal{S} = (\mathcal{T}, \mathcal{M}, \Sigma)$ where $\mathcal{T}$ is an ontology, Σ is a database schema with key dependencies, and $\mathcal{M}$ is a set of mappings between $\mathcal{T}$ and Σ. Given an OBDA specification $\mathcal{S}$ and a database instance D, we call the pair $(\mathcal{S}, D)$ an *OBDA instance*. Given an OBDA instance $\mathcal{O} = ((\mathcal{T}, \mathcal{M}, \Sigma), D)$, the *virtual RDF graph exposed by $\mathcal{O}$* is the RDF graph $\mathcal{G}^{\mathcal{A}_{\mathcal{M},D}}$; the *saturated virtual RDF graph $\mathcal{G}^{\mathcal{O}}$ exposed by $\mathcal{O}$* is the RDF graph $\mathcal{G}^{(\mathcal{T},\mathcal{A}_{\mathcal{M},D})}$.

Example 2. The mappings for the classes and properties introduced in Example 1 are:

[2] A wellbore is a three-dimensional representation of a hole in the ground.

```
:Wellbore-{wellbore_s} rdf:type :Wellbore
← SELECT wellbore_s FROM wellbore WHERE wellbore.r_existence_kd_nm = 'actual'

:Wellbore-{wellbore_s} :isInWell :Well-{well_s}
← SELECT well_s, wellbore_s FROM wellbore WHERE wellbore.r_existence_kd_nm = 'actual'

:Wellbore-{wellbore_s} :hasInterval :WellboreInterval-{wellbore_intv_s}
← SELECT wellbore_s, wellbore_intv_s FROM wellbore_interval

:Wellbore-{wellbore_s} :completionDate '{year}-{month}-{day}'^^xsd:date
← SELECT wellbore_s, year, month, day FROM wellbore WHERE wellbore.r_existence_kd_nm = 'actual'

:Wellbore-{wellbore_s} rdf:type :ProdWellbore
← SELECT w.wellbore_s AS wellbore_s FROM wellbore w, facility_clsn  WHERE complex-expression
```

Query Answering in OWL 2 QL KBs. A *conjunctive query* $q(\mathbf{x})$ is a first order formula of the form $\exists \mathbf{y}.\, \varphi(\mathbf{x}, \mathbf{y})$, where $\varphi(\mathbf{x}, \mathbf{y})$ is a conjunction of equalities and atoms of the form $A(t)$, $P(t_1, t_2)$ (where $A \in N_C, P \in N_R$), and each t, t_1, t_2 is either a *term* or an individual variable in $\mathbf{x}, \mathbf{y}$. Given a conjunctive query $q(\mathbf{x})$ and a knowledge base $\mathcal{K} := (\mathcal{T}, \mathcal{A})$, a tuple $\mathbf{i} \in N_I^{|\mathbf{x}|}$ is a *certain answer* to $q(\mathbf{x})$ iff $\mathcal{K} \models q(\mathbf{i})$. The task of query answering in OWL 2 QL (*DL-Lite*$_\mathcal{R}$) can be addressed by query rewriting techniques [4]. For an OWL 2 QL ontology $\mathcal{T}$, a conjunctive query q can be rewritten to a union q_r of conjunctive queries such that for each assertion set $\mathcal{A}$ and each tuple of individuals $\mathbf{i} \in N_I^{|\mathbf{x}|}$, it holds $(\mathcal{T}, \mathcal{A}) \models q(\mathbf{i}) \Leftrightarrow \mathcal{A} \models q_r(\mathbf{i})$. Many rewriting techniques have been proposed in the literature [14,22,3].

SPARQL [9] is a W3C standard language designed to query RDF graphs. Its vocabulary contains four pairwise disjoint and countably infinite sets of symbols: `I` for *IRIs*, `B` for *blank nodes*, `L` for *RDF literals*, and `V` for *variables*. The elements of $C = \mathtt{I} \cup \mathtt{B} \cup \mathtt{L}$ are called *RDF terms*. A *triple pattern* is an element of $(C \cup \mathtt{V}) \times \mathtt{I} \times (C \cup \mathtt{V})$. A *basic graph pattern* (*BGP*) is a finite set of joins of triple patterns. BGPs can be combined using the SPARQL operators join, optional, filter, projection, etc.

Example 3. The following SPARQL query, containing a BGP with three triple patterns, returns all the wellbores, their completion dates, and the well where they are contained.

```
SELECT * WHERE {?wlb rdf:type :Wellbore. ?wlb:completionDate ?cmpl. ?wlb:isInWell ?w.}
```

To ease the presentation of the technical development, in the rest of this paper we adopt the OWL 2 QL entailment regime for SPARQL query answering [15], but disallow complex class/property expressions in the query. Intuitively this restriction states that each BGP can be seen as a conjunctive query without existentially quantified variables. Under this restricted OWL 2 QL entailment regime, the task of answering a SPARQL query q over a knowledge base $(\mathcal{T}, \mathcal{A})$ can be reduced to answering q over the saturated graph $\mathcal{G}^{(\mathcal{T}, \mathcal{A})}$ under the simple entailment regime. This restriction can be lifted with the help of a standard query rewriting step [15].

3 SPARQL Query Answering in OBDA

In this section we describe the typical steps that an OBDA system performs to answer SPARQL queries and discuss the performance challenges. To do so, we pick the representative state-of-the-art OBDA system *Ontop* and discuss its functioning in detail.

During its start-up, *Ontop* classifies the ontology, "compiles" the ontology into the mappings generating the so-called $\mathcal{T}$-mappings [19], and removes redundant mappings by using inclusion dependencies (e.g., foreign keys) contained in the database schema. Intuitively, $\mathcal{T}$-mappings expose a saturated RDF graph. Formally, given a basic OBDA specification $\mathcal{S} = (\mathcal{T}, \mathcal{M}, \Sigma)$, the mappings $\mathcal{M}_\mathcal{T}$ are $\mathcal{T}$-mappings for $\mathcal{S}$ if, for every OBDA instance $\mathcal{O} = (\mathcal{S}, D)$, $\mathcal{G}^\mathcal{O} = \mathcal{G}^{(\mathcal{A}_{\mathcal{M}_\mathcal{T}}, D)}$.

Example 4. The $\mathcal{T}$-mappings for our running example are those in Example 2 plus

```
:Wellbore-{wellbore_s} rdf:type :Wellbore
← SELECT wellbore_s FROM wellbore WHERE wellbore.r_existence_kd_nm = 'actual'

:Wellbore-{wellbore_s} rdf:type :Wellbore
← SELECT wellbore_s, wellbore_intv_s FROM wellbore_interval

:Wellbore-{wellbore_s} rdf:type :Wellbore
← SELECT w.wellbore_s FROM wellbore w, facility_clsn  WHERE  ... complex-expression
```

The new mappings are derived from the domain of the properties `:isInWell`, `:completionDate`, and because `:ProdWellbore` is a sub-class of `:Wellbore`.

After the start-up, in the query answering stage, *Ontop* translates the input SPARQL query into an SQL query, evaluates it, and returns the answers to the end-user. We divide this stage in five phases: *(a)* the SPARQL query is *rewritten* using the tree-witness rewriting algorithm; *(b)* the rewritten SPARQL query is *unfolded* into an SQL query using $\mathcal{T}$-mappings; *(c)* the resulting SQL query is optimized; *(d)* the optimized SQL query is executed by the database engine; *(e)* the SQL result is translated into the answer to the original SPARQL query. For the sake of simplicity, we disregard phase *(a)* since it goes out of the scope of this paper (cf. [10]), and phases *(d)* and *(e)* because they are straightforward. In the following we elaborate on phases *(b)* and *(c)*.

From SPARQL to SQL. In phase *(b)* the rewritten SPARQL query is unfolded into an SQL query using $\mathcal{T}$-mappings. The rewritten query is first transformed into a tree representation of its SPARQL algebra expression. The algorithm starts by replacing each leaf of the tree, that is, a triple pattern of the form (s, p, o), with the union of the SQL queries defining p in the $\mathcal{T}$-mapping. Such SQL queries are obtained as follows: given a triple pattern $p =$ `?x rdf:type :A`, and a mapping $m =$ `:A`$(f(\mathbf{y}')) \leftarrow sql(\mathbf{y})$, the *SQL unfolding* $\text{unf}(p, m)$ *of* p *by* m is the SQL query `SELECT` $\tau(f(\mathbf{y}'))$ `AS x FROM` $sql(\mathbf{y})$, where τ is an SQL function filling the placeholders in f with values in $\mathbf{y}'$. We denote the sub-expression "`SELECT` $\tau(f(\mathbf{y}'))$ `AS x`" by $\pi_{x/f(\mathbf{y}')}$. The notions of "unf" and "π" are defined similarly for properties.

Example 5. Consider the triple pattern $p =$ `?wlb :completionDate ?d`, and the fourth mapping m from Example 2. Then the SQL unfolding $\text{unf}(p, m)$ is the SQL query

```
SELECT CONCAT(":Wellbore-",well_s) AS wlb,CONCAT("'",year,"-",month,"-", day,"'^^xsd:date") AS d
FROM wellbore WHERE wellbore.r_existence_kd_nm = 'actual'
```

Given a triple pattern p and a set of mappings $\mathcal{M}$, the *SQL unfolding* $\text{unf}(p, \mathcal{M})$ *of* p *by* $\mathcal{M}$ is the SQL union $\cup_{m \in \mathcal{M}}\{\text{unf}(p, m) \mid \text{unf}(p, m)$ *is defined*$\}$.

Once the leaves are processed, the algorithm processes the upper levels in the tree, where the SPARQL operators are translated into the corresponding SQL operators

(Project, InnerJoin, LeftJoin, Union, and Filter). Once the root is translated the process terminates and the resulting SQL expression is returned.

Example 6. The unfolded SQL query for the SPARQL query in Example 3 and $\mathcal{T}$-mappings in Example 4 has the following shape:

$$(\pi_{wlb/\square} sql_{:\texttt{Wellbore}} \cup \pi_{wlb/\square} sql_{:\texttt{ProdWellbore}} \cup \pi_{wlb/\square} sql_{:\texttt{hasInterval}}) \\ \bowtie (\pi_{wlb/\square,cmp/\diamond} sql_{:\texttt{completionDate}}) \bowtie (\pi_{wlb/\square,w/\circ} sql_{:\texttt{isInWell}})$$

where $\square$ = `:Wellbore-{wellbore_s}`, $\diamond$ = `'{year}-{month}-{day}'^^xsd:date`, $\circ$ = `:Well-{well_s}`, and sql_P is the SQL query in the mapping defining the class/property P.

Optimizing the generated SQL queries. At this point, the unfolded SQL queries are merely of theoretical value as they would not be efficiently executable by any database system. A problem comes from the fact that they contain joins over the results of built-in database functions, which are expensive to evaluate. Another problem is that the unfoldings are usually verbose, often containing thousands of unions and join operators. Structural and semantic optimizations are in charge of dealing with these two problems.

Structural Optimizations. To ease the presentation, we assume the queries to contain only one BGP. Extending to the general case is straightforward. An SQL unfolding of a BGP has the shape of a join of unions $Q = Q_1 \bowtie Q_2 \ldots \bowtie Q_n$, where each Q_i is a union of sub-queries. The first step is to remove duplicate sub-queries in each Q_i. In the second step, Q is transformed into a union of joins. In the third step, all joins of the kind $\pi_{x/f} sql_1(\mathbf{z}) \bowtie \pi_{x/g} sql_2(\mathbf{w})$ where $f \neq g$ are removed because they do not produce any answer. In the fourth step, the occurrences of the SQL function π for creating URIs are pushed to the root of the query tree so as to obtain efficient queries where the joins are over database values rather than over URIs. Finally, duplicates in the union are removed.

Semantic Optimizations. SQL queries are semantically analyzed with the goal of transforming them into a more efficient form. The analyses are based on database integrity constraints (precisely, primary and foreign keys) explicitly defined in the database schema. These constraints are used to identify and remove redundant self-joins and unions from the unfolded SQL query.

How Optimized are Optimized Queries? There are real-world cases where the optimizations discussed above are not enough to mitigate the exponential explosion caused by the unfolding. As a result, the unfolded SQL queries cannot be efficiently handled by DB engines [16]. However, the same queries can usually be manually formulated in a succint way by database managers. A reason for this is that database dependencies cannot model certain domain constraints or storage policies that are available to the database manager but not to the OBDA system. The next example, inspired by the Statoil use case explained in Section 5, illustrates this issue.

Example 7. The data stored at Statoil has certain properties that derive from domain constraints or storage policies. Consider a modified version of the query defining the class `:Wellbore` where all the attributes are projected out. According to storage policies for the database table `wellbore`, the result of the evaluation of this query against any database instance must satisfy the following constraints: *(i)* it must contain all the

wellbores[3] in the ontology (modulo templates); *(ii)* every tuple in the result must contain the information about name, date, and well (no nulls); *(iii)* for each wellbore in the result, there is exactly one date/well that is tagged as 'actual'.

Query with Redundant Unions. Consider the SPARQL query retrieving all the wellbores, namely `SELECT * WHERE {?wlb rdf:type :Wellbore.}`. By ontological reasoning, the query will retrieve also the wellbores that can be inferred from the subclasses of `:Wellbore` and from the properties where `:Wellbore` is the domain or range. Thus, after unfolding and optimizations, the resulting SQL query has the structure $\pi_{wlb/\square}(sql_1)$, with $sql_1 = (sql_{\texttt{:Wellbore}} \cup sql_{\texttt{:ProdWellbore}} \cup \pi_{\#} sql_{\texttt{:hasInterval}})$, where $\square$ = `:Wellbore-{wellbore_s}`, and `#` = `wellbore_s`. However, all the answers returned by sql_1 are also returned by the query $sql_{\texttt{:Wellbore}}$ alone, when these two queries are evaluated on a data instance satisfying item *(i)*.

Query with Redundant Joins. For the SPARQL query in Example 3, the unfolded and optimized SQL translation is of the form $\pi_{wlb/\square,cmp/\diamond,w/\circ}(sql_2)$ with $sql_2 = sql_1 \bowtie sql_{\texttt{:completionDate}} \bowtie sql_{\texttt{:isInWell}}$. Observe that the answers from sql_2 could also be retrieved from a projection and a selection over `wellbore`. This is because sql_1 could be simplified to $sql_{\texttt{:Wellbore}}$ and items *(ii)* and *(iii)*. The problem we highlight here is that this "optimized" SQL query contains two redundant joins if storage policies and domain constraints are taken into account.

It is important to remark that the constraints in the previous example cannot be expressed through schema dependencies like foreign or primary keys (because these constraints are defined over the output relations of SQL queries in the mappings, rather than over database relations[4]). Therefore, current state-of-the-art optimizations applied in OBDA cannot exploit this information.

4 OBDA Constraints

We now formalize two properties over an OBDA instance: *exact predicates* and *virtual functional dependencies*. We will then enrich the OBDA specification with a constraints component, stating that all the instances for the specification display such properties.We show how this additional constraint component can be used to identify and remove redundant unions and joins from the unfolded queries.

From now on, let $\mathcal{O} = (\mathcal{S}, D)$ be an OBDA instance of a specification $\mathcal{S} = (\mathcal{T}, \mathcal{M}, \Sigma)$.

4.1 Exact Predicates in an OBDA Instance

In real world scenarios it often happens that axioms in the ontology do not enrich the answers to queries. Often this is due to storage policies not available to the OBDA system. This fact leads to redundant unions in the generated SQL, as shown in Example 7. In this section we show how certain properties defined on the mappings and the predicates, ideally deriving from such constraints, can be used to reduce the number of redundant unions in the generated SQL queries for a given OBDA instance.

Definition 1 (Exact Mapping). *Let $\mathcal{M}'$ be a set of mappings defining a predicate A. We say that $\mathcal{M}'$ is exact for A in $\mathcal{O}$ if $\mathcal{O} \models A(\boldsymbol{a})$ if and only if $((\emptyset, \mathcal{M}', \Sigma), D) \models A(\boldsymbol{a})$.*

[3] i.e., individuals in the class `:Wellbore`

[4] Materializing the SQL in the mappings is not an option, since the schema is fixed.

In practice it is often the case that the mappings for a particular predicate declared in the OBDA specification are already exact. This leads us to the next definition.

Definition 2 (Exact Predicate). *A predicate A is exact in $\mathcal{O}$ if the set of all the mappings in $\mathcal{M}$ defining A are exact for A in $\mathcal{O}$.*

Recall that *Ontop* adds new mappings to the initial set of mappings through the $\mathcal{T}$-mapping technique. For exact predicates, this can be avoided while producing the same saturated virtual RDF graph. Fewer mappings lead to unfoldings with less unions.

Proposition 1. *Let $\mathcal{M}'$ be exact for the predicate A in $\mathcal{T}$. Let $\mathcal{M}'_{\mathcal{T}}$ be the result of replacing all the mappings defining A in $\mathcal{M}_{\mathcal{T}}$ by $\mathcal{M}'$. Then $\mathcal{G}^{\mathcal{O}} = \mathcal{G}^{((\emptyset,\mathcal{M}'_{\mathcal{T}},\Sigma),D)}$.*

Example 8. The $\mathcal{T}$-mappings for `:Wellbore` consist of four mappings (see Example 4). However, `:Wellbore` is an exact class (Example 7). Therefore we can drop the three $\mathcal{T}$-mappings for `:Wellbore` inferred from the ontology, and leave only its original mapping.

4.2 Functional Dependencies in an OBDA instance

Recall that in database theory a functional dependency (abbr. FD) is an expression of the form $\mathbf{x} \to \mathbf{y}$, read ***x** functionally determines **y***, where $\mathbf{x}$ and $\mathbf{y}$ are tuples of attributes. We say that $\boldsymbol{x} \to \boldsymbol{y}$ *is over an attributes set R* if $\mathbf{x} \subseteq R$ and $\mathbf{y} \subseteq R$. Finally, $\boldsymbol{x} \to \boldsymbol{y}$ *is satisfied by a relation I on R* if $\mathbf{x} \to \mathbf{y}$ is over R and for all tuples $\mathbf{u}, \mathbf{v} \in I$, if the value $\mathbf{u}[\mathbf{x}]$ of $\mathbf{x}$ in $\mathbf{u}$ is equal to the value $\mathbf{v}[\mathbf{x}]$ of $\mathbf{x}$ in $\mathbf{v}$, then $\mathbf{u}[\mathbf{y}] = \mathbf{v}[\mathbf{y}]$. Whenever R is clear from the context, we simply say that $\mathbf{x} \to \mathbf{y}$ is satisfied in $\mathcal{I}$.

A *virtual functional dependency* intuitively describes a functional dependency on a saturated virtual RDF graph. We identify two types of virtual functional dependencies:

- *Branching VFD*: This dependency describes the relation between an object and a set of functional properties providing information about this object. Intuitively, it corresponds to a "star" of "functional-like"[5] properties in the virtual RDF graph. For instance, given a person, the properties describing its (unique) gender, national id, biological mother, etc. are a branching VFD.
- *Path VFD*: This dependency describes the case when, from a given individual and a list of properties, there is at most one path that can be followed using the properties in the list. For instance, x works in a single department y, and y has a single manager w, and w works for a single company z.

We use these notions to identify those cases where a SPARQL join of properties translates into a redundant SQL join.

Definition 3 (Virtual Functional Dependency). *Let t be a template, and S_t be the set of individuals in $G^{\mathcal{O}}$ generated from t. Let $P, P_1, \ldots, P_n$ be properties in $\mathcal{T}$. Then*

- *A* branching VFD *is an expression of the form $t \mapsto^b P_1 \cdots P_n$. A VFD $t \mapsto^b P$* is satisfied in $\mathcal{O}$ *if for each element $s \in S_t$, there are no $o \neq o'$ in $G^{\mathcal{O}}$ such that $\{(s, P, o), (s, P, o')\} \subseteq G^{\mathcal{O}}$. A VFD $t \mapsto^b P_1 \cdots P_n$ is satisfied in $\mathcal{O}$ if $t \mapsto^b P_i$ is satisfied in $\mathcal{O}$ for each $i \in \{1, \ldots, n\}$.*

[5] A property which is functional when restricting its domain/range to individuals generated from a single template.

– *A* path VFD *is an expression of the form* $t \mapsto^p P_1 \cdots P_n$. *A VFD* $t \mapsto^p P_1 \cdots P_n$ is satisfied in $\mathcal{O}$ *if for each* $s \in S_t$ *there is at most one list of nodes* $(o_1, \ldots, o_n)$ *in* $G^{\mathcal{O}}$ *such that* $\{(s, P_1, o_1), \ldots, (o_{n-1}, P_n, o_n)\} \subseteq G^{\mathcal{O}}$.

The next example shows, similarly as in [23], that general path VFDs cannot be expressed as a combination of path VFDs of length 1.

Example 9. Let $\mathcal{G}^{\mathcal{O}} = \{(s, P_1, o_1), (o_1, P_2, o_2), (s, P_1, o_1')\}$, and t a template such that $S_t = \{s\}$. Then, $t \mapsto^p P_1P_2$ is clearly satisfied in $\mathcal{O}$. However, $t \mapsto^p P_1$ is not.

A property P might not be functional, but still $t \mapsto^b P$ might be satisfied in $\mathcal{O}$ for some t.

Example 10. Let $\mathcal{G}^{\mathcal{O}} = \{(s, P, o_1), (s, P, o_2), (s', P, o_3)\}$, and t a template such that $S_t = \{s'\}$. Then, the VFD $t \mapsto^p P$ is satisfied in $\mathcal{O}$, but P is not functional.

A functional dependency satisfied in the virtual RDF graph might not correspond to a functional dependency over the database relations. We show this with an example:

Example 11. Consider the following instance of the view `wellbore`.

wellbore_s	year	month	day	r_existence_kd_nm	well_s
002	2010	04	01	historic	1
002	2009	04	01	actual	1

The mapping defining `:completionDate` (c.f. Example 2) uses the view `wellbore` and has a filter `r_existence_kd_nm='actual'`. Observe that there is no FD (`wellbore_s` → `year month day`). However, the VFD `:Wellbore-{}` $\mapsto^b$ `:completionDate` is satisfied with this data instance, since in $\mathcal{G}^{\mathcal{O}}$ the wellbore `:Wellbore-002` is connected to a single date `"2010-04-01"^^xsd:date` through `:completionDate`.

Functional dependencies satisfied in a database instance often do not correspond to any VFD at the virtual level. We show this with an example:

Example 12. Consider the table $T_1(x, y, z)$ with a single tuple: $(1, 2, 3)$. Clearly $x \to y$ and $x \to z$ are FDs satisfied in T_1. Now consider the following mappings:

`:{x}` P_1 `:{y}` ← **SELECT** * **FROM** T_1	`:{x}` P_1 `:{z}` ← **SELECT** * **FROM** T_1

Clearly, there is no VFD involving P_1.

Hence, the shape of the mappings affects the satisfiability of VFDs. Moreover, the ontology can also affect satisfiability. We show this with an example:

Example 13. Consider again the data instance D_E from Example 12, and the mappings $\mathcal{M}_E$

`:{x}` P_1 `:{y}` ← **SELECT** * **FROM** T_1	`:{x}` P_2 `:{z}` ← **SELECT** * **FROM** T_1

Consider an OBDA instance $\mathcal{O}_E = ((\emptyset, \mathcal{M}_E, \Sigma_E)D_E)$. Then the virtual functional dependencies `:{}` $\mapsto^b P_1$ and `:{}` $\mapsto^b P_2$ are satisfied in $\mathcal{O}$. Consider another OBDA instance $\mathcal{O}'_E = ((\mathcal{T}_E, \mathcal{M}_E, \Sigma_E), D_E)$, where $\mathcal{T}_E = \{P_1$ `rdfs:subClassOf` $P_2\}$. Then the two VFDs above are not satisfied in $\mathcal{O}'_E$.

VFD Based Optimization In this section we show how to optimize queries using VFDs. Due to space limitations, we focus on branching VFDs. The results for path VFDs are analogous and can be found in the technical report [12], as well as proofs.

Definition 4. *The set of mappings $\mathcal{M}$ is* basic for $\mathcal{T}$ *if, for each property P in $\mathcal{T}$, P is defined by at most one mapping in $\mathcal{M}_{\mathcal{T}}$. We say that $\mathcal{O}$ is basic if $\mathcal{M}$ is basic for $\mathcal{T}$.*

To ease the presentation, from now on we assume $\mathcal{O}$ to be basic. We denote the (unique) mapping for P_i in $\mathcal{T}$, $i \in \{1, \dots, m\}$, as

$$t_d^i(\mathbf{x}_i) \quad P_i \quad t_r^i(\mathbf{y}_i) \leftarrow sql_i(\mathbf{z}_i).$$

where t_d^i, and t_r^i are templates for the domain and range of P_i, and $\mathbf{x}_i$, $\mathbf{y}_i$ are lists of attributes in $\mathbf{z}_i$. The list $\mathbf{z}_i$ is the list of projected attributes, which we assume to be the maximal list of attributes that can be projected from sql_i.

Although we only consider basic instances, we show in the technical report [12] how the results from this section can also be applied to the general case.

We also assume that queries $sql_i(\mathbf{z}_i)$ always contain a filter expression of the form $\sigma_{\text{notNull}(\mathbf{x}_i,\mathbf{y}_i)}$, even if we do not specify it explicitly in the examples, since URIs cannot be generated from nulls [6]. Without loss of generality, we assume that $\mathbf{z}_1$ contains all the attributes in $\mathbf{x}_1, \mathbf{y}_1, \dots, \mathbf{y}_n$.

In order to check satisfiability for a VFD in an OBDA instance one can analyze the DB based on the mappings and the ontology. The next lemma formalizes this intuition.

Lemma 1. *Let $P_1, \dots, P_n$ be properties in $\mathcal{T}$ such that, for each $1 \leq i < n$, $t_d^i = t_d^1$. Then, the VFD $t_d^1 \mapsto^b P_1 \dots P_n$ is satisfied in $\mathcal{O}$ if and only if, for each $1 \leq i \leq n$, the FD $\boldsymbol{x}_i \rightarrow \boldsymbol{y}_i$ is satisfied on $sql_i(\boldsymbol{z}_i)^D$.*

Example 14. Consider the properties `:inWell` and `:completionDate` from our running example. The lemma above suggests that the VFD `:Wellbore-{}` $\mapsto^b$ `:isInWell :completionDate` is satisfied in our OBDA instance with a database instance D if and only if *(i)* `wellbore_s`$\rightarrow$`well_s` is satisfied in $sql^D_{:isInWell}$, and *(ii)* `wellbore_s`$\rightarrow$`year month day` is satisfied in $sql^D_{:completionDate}$.

From Example 7, there is an organization constraint for the view `wellbore` forcing only one completion date for each "actual" wellbore. As a consequence, the two FDs *(i)* and *(ii)* hold in any database D following this organization constraint. Therefore, the VFD in such instance is also satisfied.

We now show how VFDs can be used to *find redundant joins* that can be eliminated in the SQL translations.

Definition 5 (Optimizing Branching VFD). *Let t be a template. An* optimizing branching VFD *is an expression of the form $t \rightsquigarrow^b P_1 \cdots P_n$. An optimizing VFD $t \rightsquigarrow^b P_1 \cdots P_n$* is satisfied in $\mathcal{O}$ *if $t \mapsto^b P_1 \cdots P_n$ is satisfied in $\mathcal{O}$, and for each $i \in \{1, \dots, n\}$ it holds*

$$\pi_{\boldsymbol{x}_1,\boldsymbol{y}_i} sql_1(\boldsymbol{z}_1)^D \subseteq \rho_{\boldsymbol{x}_1/\boldsymbol{x}_i}(\pi_{\boldsymbol{x}_i,\boldsymbol{y}_i} sql_i(\boldsymbol{z}_i))^D \tag{1}$$

Example 15. Recall that the VFD `:Wellbore-{}` $\mapsto^b$ `:isInWell, :completionDate` in Example 14 is satisfied in our OBDA instance. The precondition (1) holds because (a) the properties

are defined by the same SQL query (modulo projection) and (b) the organization constraint "each wellbore entry must contain the information about name, date, and well (no nulls)". Thus, the optimizing VFD `:Wellbore-{}` $\rightsquigarrow^b$ `:isInWell, :completionDate` is satisfied in this instance.

Lemma 2. *Consider n properties $P_1, \ldots, P_n$ with $t_d^i = t_d^1$, for each $1 \leq i \leq n$, and for which $t_d^1 \rightsquigarrow^b P_1 \cdots P_n$ is satisfied in $\mathcal{O}$. Then*

$$\pi_\gamma(sql_1(\mathbf{z}_1))^D = \pi_\gamma(sql_1(\mathbf{z}_1) \bowtie_{\boldsymbol{x}_1=\boldsymbol{x}_2} sql_2(\mathbf{z}_2) \bowtie \cdots \bowtie_{\boldsymbol{x}_1=\boldsymbol{x}_n} sql_n(\mathbf{z}_n))^D,$$

where $\gamma = \boldsymbol{x}_1, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$.

We now show how virtual functional dependencies can be used in presence of triple patterns of the form `?z rdf:type C`. As for properties, We assume that for each concept C_j we have a single $\mathcal{T}$-mapping of the form $C_j(t^j(\mathbf{x})) \leftarrow sql_j(\mathbf{z}_j)$.

Definition 6 (Domain Optimizing Class Expression). *A* domain optimizing class expression *(domain OCE) is an expression of the form $t_j \rightsquigarrow^d_{P_i} C_j$. We say that $t_j \rightsquigarrow^d_{P_i} C_j$ is satisfied in $\mathcal{O}$ if $t_j = t_d^i$ and $\pi_x sql_j(\mathbf{z}_j)^D \supseteq \rho_{x/\boldsymbol{x}_i}(\pi_{\boldsymbol{x}_i} sql_i(\mathbf{z}_i))^D$.*

Definition 7 (Range Optimizing Class Expression). *A* range optimizing class expression *(range OCE) is an expression of the form $t_j \rightsquigarrow^r_P C_j$. We say that $t_j \rightsquigarrow^r_{P_i} C_j$ is satisfied in $\mathcal{O}$ if $t_j = t_r^i$ and $\pi_{\boldsymbol{x}} sql_j(\mathbf{z}_j)^D \supseteq \rho_{\boldsymbol{x}/\boldsymbol{y}_i}(\pi_{\boldsymbol{y}_i} sql_i(\mathbf{z}_i))^D$.*

Optimizing VFDs and classes give us a tool to identify those BGPs whose SQL translation can be optimized by removing redundant joins.

Definition 8 (Optimizable branching BGP). *A BGP β is optimizable w.r.t. $\mathfrak{v} = t_d \rightsquigarrow^b P_1 \ldots P_n$ if* (i) $\mathfrak{v}$ *is satisfied in $\mathcal{O}$;* (ii) *the BGP of triple patterns in β involving properties is of the form* `?v` P_1 `?`v_1`. ...?v` P_n `?`v_n`.; and* (iii) *for each triple pattern of the form* `?u rdf:type C` *in $\beta$ ,* `?u` *is either the subject of some P_i and $t_d^i \rightsquigarrow^d_{P_i} C$ is satisfied in $\mathcal{O}$, or ?u is in the object of some P_i and $t_r^i \rightsquigarrow^r_{P_i} C$ is satisfied in $\mathcal{O}$.*

Finally, we prove that the standard SQL translation of optimizable BGPs contains redundant SQL joins that can be safely removed.

Theorem 1. *Let β be an optimizable BGP w.r.t. $t_d \rightsquigarrow^x P_1 \ldots P_n$ $(x = b, p)$ in $\mathcal{O}$. Let $\pi_{v/t_d^1, v_1/t_r^1, \ldots, v_n/t_r^n} sql_\beta$ be the SQL translation of β as explained in Section 3. Let $sql'_\beta = sql_1(\boldsymbol{x}_1, \boldsymbol{y}_1 \ldots, \boldsymbol{y}_n)$. Then sql_β^D and $sql_\beta'^D$ return the same answers.*

Corollary 1. *Let Q be a SPARQL query. Let sql_Q be the SQL translation of Q as explained in Section 3. Let sql'_Q be the SQL translation of Q where all the SQL expressions corresponding to an optimizable BGPs w.r.t. a set of VFDs have been optimized as stated in Theorem 1. Then sql_Q^D and $sql_Q'^D$ return the same answers.*

Example 16. It is clear that the class `:Wellbore` is optimizing w.r.t. the domain of `:completionDate` and `:isInWell`. Since `:Wellbore-{}` $\rightsquigarrow^b$ `:completionDate, :isInWell` is satisfied (c.f. Example 15), one can allow the semantic optimizations to safely remove redundant joins in query sql_1, sketched in Example 7. From Theorem 1, it follows that, $sql_{:Wellbore} \bowtie sql_{:completionDate} \bowtie sql_{:isInWell}$ can be by simplified to $sql_{:Wellbore}$.

4.3 Enriching the OBDA Specification with Constraints

We propose to enrich the traditional OBDA specification with a constraint component, so as to allow the OBDA system to perform enhanced optimization as described in the previous section. More formally, an *OBDA specification with constraints* is a tuple $\mathcal{S}_{constr} = (\mathcal{S}, C)$ where $\mathcal{S}$ is an OBDA specification and C is a set of exact mappings, exact predicates, optimizing virtual functional dependencies, and optimizing class expressions. An *instance of* $\mathcal{S}_{constr}$ is an OBDA instance of $\mathcal{S}$ satisfying the constraints in C. Our intention is to be able to use more of the constraints that exist in real databases for query optimization, since we often see that these cannot be expressed by existing database constraints (i.e. keys). Since $\mathcal{S}$ does not necessarily imply C, checking the validity of C may have to take into account more information than just S. The constraints C may be known to hold e.g. by policy, or be enforced by external tools, e.g., as in the case mentioned in the experiments below, by the tool used to enter data into the database.

In order to aid the user in the specification of C, we implemented tools to identify what exact mappings and optimizing virtual functional dependencies are satisfied in a given OBDA instance (see [12]). . The user can then verify whether these suggested constraints hold in general, for example because they derive from storage policies or domain knowledge, and provide them as parameters to the OBDA system. The user intervention is necessary, because constraints derived from actual data can be an artifact of the current situation of the database.

Optimizing VFD Constraints. We have implemented a tool that automatically finds a restricted type of optimizing VFDs satisfied in a given OBDA instance and we have extended *Ontop* to complement semantic optimization using these VFDs. This implementation aims to mitigate the problem of redundant self-joins resulting from reifying relational tables. Although this is a simple case, it is extremely common in practice and, as we show in our experiments in Section 5, this class of VFDs is powerful enough to sensibly improve the execution times in real world scenarios.

Exact Predicates Constraints. We implemented a tool to find exact predicates, and we extended *Ontop* to optimize $\mathcal{T}$-mappings with them. For each predicate P in the ontology $\mathcal{T}$ of an OBDA instance $\mathcal{O}$, the tool constructs the query q that returns all the individual/pairs in P. Then it evaluates q in the two OBDA instances $\mathcal{O}$ and $((\emptyset, \mathcal{M}, \Sigma), D)$. If the answers for q coincide in both instances, then P is exact.

5 Experiments

In this section we present a set of experiments evaluating the techniques described above. In [12] we ran additional controlled experiments using an OBDA benchmark built on top of the Wisconsin benchmark [7], and obtain similar results to the ones here.

Statoil Scenario In this section we briefly describe the Statoil use-case, and the challenges it presents for OBDA. At Statoil, users access several databases on a daily basis, and one of the most important ones is the Exploration and Production Data Store (EPDS) database. EPDS is a large legacy SQL (Oracle 10g) database comprising over 1500 tables (some of them with up to 10 million tuples) and 1600 views. The complexity of the SQL schema of EPDS is such that it is counter-productive and error-prone to manually write queries over the relational database. Thus, end-users either use only a set of tools with predefined SQL queries to access the database, or interact with IT experts

Table 1: Results from the tests over EPDS.

	std. opt.	w/VFD	w/exact predicates	w/both
Number of queries timing-out	17	10	11	4
Number of fully answered queries	43	50	49	56
Avg. SQL query length (in characters)	51521	28112	32364	8954
Average unfolding time	3.929 s	3.917 s	1.142 s	0.026 s
Average total query exec. time with timeouts	376.540 s	243.935 s	267.863 s	147.248 s
Median total query exec. time with timeouts	35.241 s	11.135 s	21.602 s	14.936 s
Average successful query exec. time (without timeouts)	36.540 s	43.935 s	51.217 s	67.248 s
Median successful query exec. time (without timeouts)	12.551 s	8.277 s	12.437 s	12.955 s
Average number of unions in generated SQL	6.3	3.4	5.1	2.2
Average number of tables joined per union in generated SQL	21.0	18.2	20.0	14.2
Average total number of tables in generated SQL	132.7	62.0	102.2	31.4

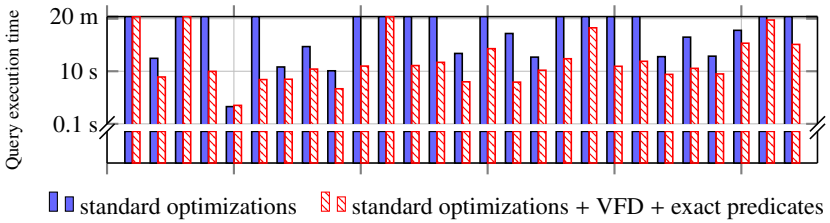

Fig. 1: Comparison of query execution time with standard optimizations.Log. scale

so as to formulate the right query. The latter process can take weeks. This situation triggered the introduction of OBDA in Statoil in the context of the Optique project [13]. In order to test OBDA at Statoil, the users provided 60 queries (in natural language) that are relevant to their job, and that cannot be easily performed or formulated at the moment. The Optique partners formulated these queries in SPARQL, and handcrafted an ontology, and a set of mappings connecting EPDS to the ontology. The ontology contains 90 classes, 37 object properties, and 31 data properties; and there are more than 140 mappings. The queries have between 0 to 2 complex filter expressions (with several arithmetic and string operations), 0 to 5 nested optionals, modifiers such as ORDER BY and DISTINCT, and up to 32 joins.

Experiment Results. The queries were executed sequentially on a HP ProLiant server with 24 Intel Xeon CPUs (X5650 @ 2.67 GHz), 283 GB of RAM. Each query was evaluated three times and we took the average. We ran the experiments with 4 exact concepts and 15 virtual functional dependencies, found with our tools and validated by database experts. The 60 SPARQL queries have been executed over *Ontop* with and without the optimizations for exact predicates and virtual functional dependencies. We consider that a query times out if the average execution time is greater than 20 minutes.

The results are summarized in Table 1 and Figure 1. We can see that the proposed optimizations allow *Ontop* to critically reduce the query size and improve the performance of the query execution by orders of magnitude. Specifically, in Figure 1 we compare standard optimizations with and without the techniques presented here. Observe that the average successful query execution time is higher with new optimizations than without because the number of successfully executed queries increases. With standard optimizations, 17 SPARQL queries time out. With both novel optimizations enabled, only four queries still time out.

A total of 27 SPARQL queries get a more compact SQL translation with new optimizations enabled. The largest proportional decrease in size of the SQL query is 94%, from 171k chars, to 10k. The largest absolute decrease in size of the SQL is 408k chars. Note that the number of unions in the SQL may decrease also only with VFD-based optimization. Since the VFD-based optimization removes joins, more unions may become equivalent and are therefore removed. The maximum measured decrease in execution time is on a query that times out with standard optimizations, but uses 3.7 seconds with new optimizations.

6 Related work

Dependencies have been intensively studied in the context of traditional relational databases [2]. Our work is related to the one in [23]; in particular their notion of path functional dependency is close to the notion of path VFD presented here. However, they do not consider neither ontologies, nor databases, and their dependencies are not meant to be used to optimize queries. There are a number of studies on functional dependencies in RDF [24,11], but as shown in Example 12, functional dependencies in RDF do not necessarily correspond to a VFD (when considering the ontology). Besides, these works do not tackle the issue of SQL query optimization.

The notion of *perfect mapping* [8] is strongly related to the notion of exact mapping. However there is a substantial difference: a perfect mapping must be *entailed* by the OBDA specification, whereas exact mappings are additional constraints that enrich the OBDA specification. For instance, perfect mappings would not be effective in the Statoil use case, where organizational constraints and storage policies are not entailed by the OBDA specification. The notion of *EBox* [21,17] was proposed as an attempt to include constraints in OBDA. However, EBox axioms are defined through a $\mathcal{T}$-box like syntax. These axioms cannot express constraints based on templates like virtual functional dependencies.

7 Conclusions

In this work we presented two novel optimization techniques for OBDA that complement standard optimizations in the area, and enable efficient SPARQL query answering over enterprise relational data. We provided theoretical foundations for these techniques based on two novel OBDA constraints: virtual functional dependencies, and exact predicates. We implemented these techniques in our OBDA system *Ontop* and empirically showed their effectiveness through extensive experiments that display improvements on the query execution time up to orders of magnitude.

Acknowledgement. This work is partially supported by the EU under IP project Optique (*Scalable End-user Access to Big Data*), grant agreement n. FP7-318338.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2nd edition, 2007.
2. C. Beeri and M. Y. Vardi. The implication problem for data dependencies. In *Proceedings of the Eighth Colloquium on Automata, Languages and Programming (ICALP 1981)*, volume

115 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 1981.
3. M. Bienvenu, M. Ortiz, M. Simkus, and G. Xiao. Tractable queries for lightweight description logics. In *Proceedings of the Twentythird International Joint Conference on Artificial Intelligence (IJCAI 2013)*. IJCAI/AAAI, 2013.
4. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.
5. U. S. Chakravarthy, D. H. Fishman, and J. Minker. Semantic query optimization in expert systems and database systems. In *Proc. of DEXA*, pages 659–674, 1986.
6. S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF mapping language. W3C Recommendation, World Wide Web Consortium, Sept. 2012. Available at `http://www.w3.org/TR/r2rml/`.
7. D. J. DeWitt. The wisconsin benchmark: Past, present, and future. In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.
8. F. Di Pinto, D. Lembo, M. Lenzerini, R. Mancini, A. Poggi, R. Rosati, M. Ruzzi, and D. F. Savo. Optimizing query rewriting in ontology-based data access. In *Proceedings of the Sixteenth International Conference on Extending Database Technology (EDBT 2013)*, pages 561–572. ACM Press, 2013.
9. B. Glimm and C. Ogbuji. SPARQL 1.1 entailment regimes. W3C Recommendation, World Wide Web Consortium, Mar. 2013. Available at `http://www.w3.org/TR/sparql11-entailment/`.
10. G. Gottlob, S. Kikot, R. Kontchakov, V. V. Podolskii, T. Schwentick, and M. Zakharyaschev. The price of query rewriting in ontology-based data access. *Artificial Intelligence*, 213:42–59, 2014.
11. B. He, L. Zou, and D. Zhao. Using conditional functional dependency to discover abnormal data in RDF graphs. In *Proc. of SWIM*, pages 43:1–43:7. ACM, 2014.
12. D. Hovland, D. Lanti, M. Rezk, and G. Xiao. OBDA constraints for effective query answering (extended version). CoRR Technical Report abs/1605.04263, arXiv.org e-Print archive, 2016. Available at `http://arxiv.org/abs/1605.04263`.
13. S. Kikot, R. Kontchakov, V. V. Podolskii, and M. Zakharyaschev. Exponential lower bounds and separation for query rewriting. In *Proceedings of the Thirtyninth International Colloquium on Automata, Languages and Programming (ICALP 2012)*, pages 263–274. Springer, 2012.
14. S. Kikot, R. Kontchakov, and M. Zakharyaschev. Conjunctive query answering with OWL 2 QL. In *Proceedings of the Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2012)*, pages 275–285, 2012.
15. R. Kontchakov, M. Rezk, M. Rodriguez-Muro, G. Xiao, and M. Zakharyaschev. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In *Proceedings of the Thirteenth International Semantic Web Conference (ISWC 2014)*, volume 8796 of *Lecture Notes in Computer Science*, pages 552–567. Springer, 2014.
16. D. Lanti, M. Rezk, G. Xiao, and D. Calvanese. The NPD benchmark: Reality check for OBDA systems. In *Proceedings of the Eighteenth International Conference on Extending Database Technology (EDBT 2015)*, 2015.
17. J. Mora, R. Rosati, and O. Corcho. kyrie2: Query rewriting under extensional constraints in elhio. In *Proceedings of the Thirteenth International Semantic Web Conference (ISWC 2014)*, pages 568–583, 2014.
18. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *Journal on Data Semantics*, X:133–173, 2008.
19. M. Rodriguez-Muro, R. Kontchakov, and M. Zakharyaschev. Ontology-based data access: Ontop of databases. In *Proceedings of the Twelfth International Semantic Web Conference (ISWC 2013)*, volume 8218 of *Lecture Notes in Computer Science*, pages 558–573. Springer, 2013.

20. M. Rodriguez-Muro and M. Rezk. Efficient SPARQL-to-SQL with R2RML mappings. *Journal of Web Semantics*, 2015.
21. R. Rosati. Prexto: Query rewriting under extensional constraints in *DL-Lite*. In *Proceedings of the Ninth Extended Semantic Web Conference (ESWC 2012)*, pages 360–374, 2012.
22. R. Rosati and A. Almatelli. Improving query answering over *DL-Lite* ontologies. In *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010)*, pages 290–300, 2010.
23. G. E. Weddell. Reasoning about functional dependencies generalized for semantic data models. *ACM Trans. Database Syst.*, 17(1):32–64, Mar. 1992.
24. Y. Yu and J. Heflin. Extending functional dependency to detect abnormal data in RDF graphs. In *Proc. of ISWC*, volume 7031, pages 794–809. Springer, October 2011.