Ontology-based Data Federation

Zhenzhen Gu
Free University of Bozen-Bolzano
Bolzano, Italy

Davide Lanti
Free University of Bozen-Bolzano
Bolzano, Italy

Alessandro Mosca
Free University of Bozen-Bolzano
Bolzano, Italy

Guohui Xiao
University of Bergen
Bergen, Norway

Jing Xiong
Free University of Bozen-Bolzano
Bolzano, Italy

Diego Calvanese
Free University of Bozen-Bolzano
Bolzano, Italy

ABSTRACT
Ontology-based data access (OBDA) is a well-established approach to information management which facilitates the access to a (single) relational data source through the mediation of a high-level ontology, and the use of a declarative mapping linking the data layer to the ontology. We formally introduce here the notion of ontology-based data federation (OBDF) to denote a framework that combines OBDA with a data federation layer where multiple, possibly heterogeneous sources are virtually exposed as a single relational database. We discuss opportunities and challenges of OBDF, and provide techniques to deliver efficient query answering in an OBDF setting. Such techniques are validated through an extensive experimental evaluation based on the Berlin SPARQL Benchmark.

CCS CONCEPTS
• Information systems → Data federation tools.

KEYWORDS
OBDA, Data federation, Query optimization

1 INTRODUCTION
Ontology-based data access (OBDA) [10, 33, 42] is a well-established paradigm for querying data sources via a mediating ontology that has been successfully applied in many different domains [43]. In OBDA, the ontology is expressed in a lightweight conceptual language, such as OWL 2 QL [30], which has its formal foundations in the description logics of the DL-Lite family [12]. Typically, it is assumed that the underlying data are stored in a single relational data source, to which the ontology elements are mapped in a declarative way. Specifically, in each mapping, a SQL query over the source is mapped to a class / property of the ontology, specifying how the data retrieved from the database (DB) should be used to create instances and values that populate the class / property.

Notably, for query answering, OBDA follows a virtual approach, i.e., the data are not actually extracted from the source to populate the classes and properties, but instead a SPARQL query posed over the ontology is transformed on-the-fly into a SQL query over the data source. Such transformation takes into account both the ontology axioms (in what is generally called a rewriting step [12]) and the mappings (in an unfolding step [33, 34]), and typically may lead to a substantial blow-up in the size of the resulting SQL query w.r.t. the size of the original query. Due to this, sophisticated optimization techniques have been proposed and implemented in commercial and open source OBDA systems [10, 11, 38, 45]. Such techniques exploit the available information about constraints in the data source (e.g., primary and foreign keys), the form of the mappings, and the structure of the query in order to optimize the SPARQL-to-SQL query translation process and generate a final query that is not only as compact as possible but also efficient to execute [34, 44].

So far, such techniques have been tailored towards optimizing queries that are executed over a single data source to which the OBDA system is mapped. In many settings, however, there is the need to virtually access multiple, possibly heterogeneous, data sources in an integrated way. In this case, one can resort to data federation [20, 39], where multiple autonomous data sources are exposed transparently as a unified federated relational schema, usually called virtual database. Data federation is an active research area which has been extensively studied over the years, and many mature and highly-optimized data federation tools are currently available, both in the database community and in the Semantic Web community [19].

Data federation tools can be naturally used in combination with OBDA systems, by accessing them as if they were a single relational data source\(^1\). However, to the best of our knowledge, in current OBDA systems no provision is taken for the optimization of the generated SQL query to account for the fact that the evaluation of

\(^1\)See, e.g., https://ontop-vkg.org/tutorial/federation/.
a SQL query in a data federation system is fundamentally different from query evaluation by a standard relational DBMS engine. In this work, we consider specifically this issue and provide the following contributions to this novel setting that we call Ontology-based Data Federation (OBDF for short):

- We provide a formalization of OBDF systems, where a collection of multiple, possibly heterogeneous, federated data sources are accessed via mappings from an ontology (which captures the domain vocabulary and domain knowledge).
- We study the problem of query optimization in OBDF, by devising a set of optimization techniques specifically tailored towards federated data sources.
- We carry out an experimental evaluation over an adaptation of the Berlin SPARQL Benchmark (BSBM) [5] to the federation setting, in which we assess the effectiveness of the proposed optimization techniques.

2 PRELIMINARIES

We introduce technical preliminaries and notation that we will adopt throughout the remainder of this paper.

Relational Algebra. We assume the reader to be familiar with fundamental notions of relational algebra. As conventions, we use \( \Sigma \) to denote a (relational) DB schema, \( D \) to denote an instance of a DB schema, and \( \text{sig}(A) \) to denote the signature of a relational algebra expression \( A \), which consists of the tuple \( (a_1, \ldots, a_n) \) of attributes of the relation generated by \( A \). When we want to make the signature explicit, we use the notation \( \text{Ans}(a_1, \ldots, a_n) \). We introduce the abbreviation \( \pi_{r_1/a_1, \ldots, r_n/a_n} \) for the combination \( r_1/a_1, \ldots, r_n/a_n \) of projection and renaming, \( \theta_R \) to denote the empty relation of signature \( \text{sig}(R) \), and \( \Omega_R \) to denote a tuple of null values with signature \( \text{sig}(R) \).

Given a database instance \( D \) of \( \Sigma \) and an algebra expression \( A \), \( \text{ans}(A, D) \) denotes the set of answers of \( A \) over \( D \). Given two expressions \( A \) and \( B \) over \( \Sigma \), \( A \) is contained in \( B \), denoted as \( A \subseteq B \), if \( \text{ans}(A, D) \subseteq \text{ans}(B, D) \) for every DB instance \( D \) of \( \Sigma \). \( A \) and \( B \) are equivalent, denoted as \( A \equiv B \), if \( A \subseteq B \) and \( B \subseteq A \).

Ontology-based Data Access (OBDA). We rely here on the classic framework from [42]. Due to space limitations, we assume the reader to be familiar with ontologies and Description Logics notation, and refer to the extensive literature on the subject [4].

An OBDA specification \( O \) is a triple \( (T, M, \Sigma) \), where \( T \) is an ontology consisting of class inclusion axioms \( B \subseteq C \) and role inclusion axioms \( S \subseteq T; \Sigma \) is a relational DB schema; and \( M \) is a set of OBDA-mappings of the form \( A \triangleright \triangleright C(f(a)) \) or \( A \triangleright \triangleright P(f(a), g(b)) \), where \( A \) is a relational algebra expression over \( \Sigma \), and \( C \) and \( P \) are respectively a class and a property name of \( T \), and \( f(a) \) and \( g(b) \) are \((R\&RML)\) IRI templates [14], specifying how DB values are transformed into IRIs and RDF literals, respectively making use of the sets \( a \) and \( b \) of attributes in \( \text{sig}(A) \).

For the semantics of an OBDA instance, we refer to [33]. Intuitively, an OBDA instance exposes a (virtual) RDF graph that can be queried through SPARQL [22]. The graph is virtual in the sense that RDF triples are not materialized. Instead, to answer a SPARQL query, the query is translated on-the-fly into an equivalent SQL query over the database, called its translation.

Unfolding. In OBDA, the process of producing a SQL translation \( q \) for a SPARQL query \( Q \) over an OBDA specification \( (T, M, \Sigma) \) is called unfolding [33]. Different unfolding procedures have been proposed in the literature. For this work, we focus on two variants: the classical one aiming at producing a union of conjunctive queries (UCQ) [33], and the one aiming at producing a join of union of conjunctive queries (JUCQ) [7, 27]. One can switch from one variant to another by applying algebra transformations, specifically by pushing the joins at the bottom of the algebra-tree (UCQ) or the unions (JUCQ) through the distributive rule. We describe here the JUCQ variant, which we call unfold\(_\text{wrap} \), by means of an example. As we will see in Section 5, unfold\(_\text{wrap} \) provides the basis for the unfolding algorithm in our federated setting. Consider the following SPARQL query, asking for information about stars:

\[
\text{SELECT } \exists \text{mag WHERE } ( \exists \text{rdf:type :Star} \text{, hasMagnitude } \text{mag}) .
\]

The query above corresponds to the following SPARQL algebra tree:

```
PROJECT
JOIN
\text{SELECT } \exists \text{mag WHERE } ( \exists \text{rdf:type :Star} \text{, hasMagnitude } \text{mag}) .
```

Assume the following mappings over a DB, providing SQL definitions for the class :Star and the data property :hasMagnitude.

\[
\text{SELECT id FROM dwarf_star } \mapsto \text{:star/(id) rdf:type :Star} .
\]

\[
\text{SELECT id FROM giant_star } \mapsto \text{:star/(id) rdf:type :Star} .
\]

\[
\text{SELECT id, mag FROM star_data } \mapsto \text{:star/(id) hasMagnitude } (\text{mag}) .
\]

The unfold\(_\text{wrap} \) algorithm traverses the algebra tree in a bottom-up fashion. It starts by replacing each leaf of the tree, that is, a triple pattern of the form \((s, p, o)\), with the union of the corresponding SQL definitions in the mappings. In this step, the algorithm includes in the SQL query a function (usually implemented as string concatenation) which constructs the IRIs for \( s \) and \( o \) as specified by the IRI templates in the mappings. Once it finishes processing the leaves, the algorithm continues to the upper levels in the tree, where the SPARQL operators (PROJECT, JOIN, OPTIONAL, UNION, and FILTER) are translated into the corresponding SQL operators (Project, InnerJoin, LeftJoin, Union, and Filter, respectively). Once the root is translated, we have obtained an intermediate SQL query that is already a (non-optimized) translation. For our example:

```
PROJECT
JOIN
\text{SELECT CONCAT(':star\', id) FROM dwarf_star .}
\text{SELECT CONCAT(':star\', id), mag FROM star_data .}
```
the result of CONCAT operations that lead to different results, e.g., because of different prefixes), then that join is trivially empty and gets eliminated from the algebra tree.

In our example, at the end of the unfolding procedure unfold\_wrap we obtain the following query (expressed in relational algebra):

\[ \pi_{\text{star}(id), \text{mag}}(\pi_{\text{id}}(\text{dwarf\_star} \cup \text{giant\_star})) \pi_{\text{id}, \text{id}}(\pi_{\text{id}}(\text{star\_data})) \]

where \( \text{star}(id) \) denotes the application of the IRI template CONCAT(\( \text{'':star/'',id} \)).

**Data Federation.** Federating multiple, possibly heterogeneous data sources consists in exposing a unified view of such sources, usually called virtual database (VDB). In this paper, a data source, denoted by \( S \), can be an RDB, a NoSQL DB, or of some other type. Consider a set \( S = \{ S_1, \ldots, S_n \} \) of sources to be federated, and a function (given implicitly with \( S \)) transforming the (possibly, non-relational) schema of each source \( S_i \) into a corresponding relational schema \( \Sigma_i \). Then, the federated VDB schema (for \( S \)) is the disjoint union \( \Sigma = \bigcup_{i=1}^{n} \Sigma_i \). In the following, we use letters \( T, U \) to denote database tables, and the subscript \( i \) (e.g., \( T_i \)) to indicate that \( S_i \) is the source of table \( T \). Additionally, given an arbitrary relational algebra expression \( A \), \( \text{src}(A) \) denotes the set of sources of the atoms in \( A \), and \( \text{occ}(S_i, A) \) denotes the total number of occurrences in \( A \) of relations from \( S_i \). A data federation instance \( D \) for \( \Sigma \) is the relational instance \( (\bigcup_i \text{Id}_i \text{D}_i) \) made of the union of all instances of the (relational) schema sources in \( \Sigma \). Hence, given a query \( q \), \( \text{ans}(q, D) \) denotes the set of answers of \( q \) evaluated over the federation instance \( D \).

**Data Federation vs Data Integration.** Real-world data federation systems often provide data integration capabilities, allowing users to specify an arbitrary VDB schema integrating the schemas of the various sources. The link between such a VDB schema and the schemas of the sources is realized through GAV, LAV, or GLAV mappings [15]. In this work, we always assume a federated VDB schema. The reason is that, in our setting, the integration is performed at the level of the ontology, by exploiting the definitions provided in the OBDA mappings (that can be interpreted, in fact, as GAV mappings coming from the context of data integration).

**Local Operations vs Federated Operations.** To compute the answers for a federated query, a data federation system can delegate operations (e.g., joins and unions) to the data sources, or perform the operations itself. In this paper, we distinguish between local operations (e.g., joins performed within a data source) and federated operations (e.g., joins across multiple sources, that have to be handled at the level of the federation system).

### 3 ONTOLOGY-BASED DATA FEDERATION

Our first contribution is the definition of a general framework for enriching OBDA with data federation capabilities.

**Definition 3.1 (OBDF).** Given an ontology \( T \), a federated VDB schema \( \Sigma \), and a set \( M \) of mappings from \( \Sigma \) to \( T \), an ontology-based data federation (OBDF) specification is the OBDA specification \( \mathcal{F} = (T, M, \Sigma) \).

Hence, the notions of OBDF instance and answers to a query over an OBDF instance coincide with their OBDA counterpart. Figure 1 depicts the full process of query answering in an OBDF scenario. A federation engine (e.g., Teiid, Denodo, or Dremio) is responsible for the federation of the data sources, and an OBDA system, in this case Ontop, interacts with the federation engine as it would normally do with a single relational database.

**Opportunities and Challenges.** In line with the FAIR principles\(^2\), OBDA allows users to publish data according while complying to shared, agreed-upon vocabularies which enable interoperability between different applications. Furthermore, the ontology constitutes both a documentation about the data and a basis for enabling reasoning-based services, such as query answering w.r.t. the ontology. The added value of data federation is to extend the OBDA paradigm to multiple, possibly non-relational sources.

While benefitting from both OBDA and data federation, OBDF combines the challenges of both. Next example shows possible issues that might arise from a naive implementation of OBDF.

**Example 3.2.** Consider an enterprise, whose data is spread across different sources \( S = \{ S_1, \ldots, S_4 \} \) that need to be integrated. Consider an OBDF specification \( \mathcal{F} = (T, M, \Sigma) \), with \( T \) and \( M \) as in Figure 2, where each relation in \( M \) has a subscript \( i \) denoting the source \( S_i \) relative to that relation. For the SPARQL query \( Q \) in Figure 2, asking for products’ and inspectors’ information, the unfolding procedure would produce the SQL query \( q \) from the same figure. Observe that this query is already verbose, with 3 federated joins and 4 federated unions across the different sources. At this point, state-of-the-art OBDA systems typically apply structural optimizations transforming the JUCQ \( q \) into a UCQ, by pushing the join operators at the bottom level of the algebra tree. After this transformation, the reader can easily verify that the query obtained would consists of \( 2^4 = 16 \) unions of CQs, where each CQ has 3 join operators, thus amounting to 48 joins in total. Hence, transforming JUCQs into UCQs blindly can largely increase the number of federated, thus inefficient, operations.

To complicate the picture, it is often the case that certain relations hold across the different sources: for instance, relation PerfInfo might contain the names of all the employees in the enterprise, rendering the last union in \( q \) redundant. Similarly, the intersection of ConvenienceGoods and ShoppingGoods might be empty, rendering all joins between these two relations empty as well.

In the remainder of this work we discuss a novel unfolding procedure specific to the OBDF setting, able to choose the best strategy between UCQ and JUCQ unfoldings and to exploit relations holding across different data sources.

\(^2\) [https://www.go-fair.org/fair-principles/](https://www.go-fair.org/fair-principles/)
4 DATA HINTS

Under the standard formalization of OBDA [33], in which every IRI term behave like an injective R2RML template, the assumption that the ontology language used is OWL 2 QL, and that query answering is performed under the SPARQL 1.1 entailment regime, it is possible to determine a-priori all the joins between relations that can occur in the SQL translation of a user query [27]. This can be done by an offline analysis of the OBDA specification, that is, by collecting pairs of atoms with compatible templates. In [27], such intuition is used to collect specific statistics about the OBDA instance, with the goal of improving the performance of query answering. We adopt here a similar approach, by introducing different kinds of meta-information, called data hints (or, simply, hints), that we use to optimize query answering in OBDF.

We identify three kinds of hints: empty federated joins, containment redundancy, and materialized views. The first kind of hint, empty federated join, annotates which joins are expected to be empty when evaluated over the current data federation instance. For the definitions in this section, we assume a fixed federated VDB schema $\Sigma_S$.

Definition 4.1 (Hint 1: Empty Federated Join). Given an instance $D$ of $\Sigma_S$ and a federated join expression $FJ$ over $\Sigma_S$, we say that $FJ$ is an empty federated join w.r.t. $D$, denoted as $FJ \rightarrow_0 D$, if $\text{ans}(FJ, D) = \emptyset$.

The second kind of hint, containment redundancy, annotates the presence of redundancy (typically across different data sources).

Definition 4.2 (Hint 2: Containment Redundancy). Given an instance $D$ of $\Sigma_S$ and two expressions $A$ and $B$ over $\Sigma_S$, we say that $A$ is data-contained in $B$, denoted as $A \subseteq D$, if $\text{ans}(A, D) \subseteq \text{ans}(B, D)$. We use $A \equiv D$ to indicate that $A \subseteq D$ and $D \subseteq A$.

Materialized views [1, 21] can improve the overall performance of query answering. The ability to specify materialized views is provided by a few data federation systems, such as Teid or Dremio. In our formalization, we assume the presence of an extra source to store the materialization of the views, where such sources could be a federation system itself. This is motivated by the fact that it is often impossible or impractical to store the views directly in the sources, due to access policies, source ownership, etc.

Definition 4.3 (Hint 3: VDB Schema with Views). Let $M$ be a set of view definitions. We denote by $\Sigma_M$ the VDB schema $\Sigma_S \cup \Sigma_M$, where $\Sigma_M$ is the relational schema of a special data source $\Sigma^M$ materializing the views defined in $M$.

Consequently, an instance $D^M$ of $\Sigma^M$ is a VDB instance $D \cup D^M$ such that $D$ is an instance of $\Sigma_S$ and $D^M$ is an instance of $\Sigma_M$ conforming to the view definitions in $M$.

Finally, we assume two labeling functions: the first one characterizes whether a source is efficient or inefficient when answering queries, whereas the second one characterizes whether the data source is dynamic (i.e., its content is expected to change frequently), or static (i.e., its content is not expected to change).

5 QUERY OPTIMIZATION IN OBDF

We now discuss our solution to optimize SQL translations of SPARQL queries posed over an OBDF system. The main intuition is that, in OBDF, the ontology and mappings contain information to guide the discovery of the data hints discussed in the previous section. The overall method consists of two parts: 1) an offline hints pre-computation part, and 2) an on-line translation optimization part. For the remainder of this section, we assume a fixed OBDF specification $F = (T, M, \Sigma_S)$.

5.1 Pre-Computation of Hints

We exploit the mappings and the ontology in an OBDF specification to guide the gathering of hints. Basically, we will enumerate in advance all possible SQL joins and unions between pairs of relations that the system can possibly produce during the unfolding.

Analyzing the mappings. We start by showing how the analysis is carried out for joins. Trivially, join conditions appearing in the source part of mapping assertions give an indication on what columns of the DB can be joined. Other joins are the result of the translation of a SPARQL join into a SQL join. As proposed in [27, 28], in order to determine these we can analyze the IRI templates appearing in the mapping, and pre-compute all pairs of compatible templates (i.e., templates that generate the same IRI for some DB instance). For instance, consider the following mapping assertions:

$m_1 : T_1(a) \leadsto A(f(a)), \quad m_2 : T_2(b, c) \leadsto P(f(b), g(c))$

and the SPARQL query $\text{SELECT} \ ?x \ ?y \ \text{WHERE} \ (\exists x \ A \wedge P \ ?y)$, retrieving $A$-individuals and their $P$-successors. During unfold wrap execution, we reach the following intermediate translation:

$(\pi_{f(a)}(T_1(a) \bowtie \pi_{f(a)}(T_2(b, c)))) \cup (\pi_{f(a)}(T_1(a) \bowtie \pi_{f(b)}(T_1(d, e))))$

Note that the second join expression can be only empty, since the join condition $f(a) = h(d)$ can never be satisfied for any instantiation of $a$ and $d$. Hence, unfold wrap will remove this empty join, simplify the join between IRIs with the same template into a join between the underlying database attributes, and finally return:

$\pi_{f(a)}(T_1(a) \bowtie \pi_{f(b)}(T_2(b, c)))$

A similar analysis can be carried out for unions. For instance, consider the following additional mapping:

$m_4 : \pi_{f(b)}(T_2(b, c)) \leadsto A(f(b))$
Algorithm 1: hintify(((T, M, ΣE), D), Joins, Unions)

Output: An OBDF instance ((T, M, ΣE), D) with materialized views M, a set E of empty join hints, a set C of containment redundancy hints.

1. foreach j in joins do
   2. if j = ∅ then
      3. E ← E ∪ {j = ∅}
   4. else if j is a federated join or it is over an inefficient source then
      5. if j is static for each k ∈ src(j) then
         6. M ← M ∪ { Ebola(j → j)}
   7. foreach A ∪ B in Unions do
      8. if A ⊆ B then
         9. C ← C ∪ (A ⊆ B)
      10. if B ⊆ A then
         11. C ← C ∪ (B ⊆ A)
   12. return ((T, M, ΣE), D), E, C

Now, the unfolding for the same query becomes:

$$\pi_{a}\left(g\left(e\right)\right)\left(\left(T_{1}\left(a\right) \cup \pi_{b/a}\left(T_{2}\left(b, c\right)\right)\right) \vDash_{ab} T_{2}\left(b, c\right)\right)$$

Again, the union is between SQL columns and not over the result of applying the templates to DB values only because the templates are compatible. The same considerations apply to the case where the SPARQL query itself contains a union operation (e.g., a query retrieving all individuals in A union all those having a P-successor).

The role of the ontology. Ontology axioms increase the number of unions. Consider again the mappings m1, m2 above, and an ontology axiom \( \exists P \subseteq A \), stating that the domain of property P is class A. Then, our SPARQL query will unfold into the last SQL query from the previous paragraph. In fact, it is well-known [35] that answering SPARQL queries over an OWL 2 QL ontology and a set M of mappings can be reduced to the problem of answering SPARQL queries over an empty ontology and an enriched set of mappings, called T-mapping, obtained by compiling the ontology into M. In our example, it turns out that the set \( \{m_{1}, m_{2}, m_{4}\} \) of mappings is a T-mapping compiling the axiom \( \exists P \subseteq A \) into \( \{m_{1}, m_{2}\} \).

Constructing the hints. Summing up, under the assumption of using unfold\_wrap for the translation, we derive the following observations: (1) the enumeration of possible joins between DB columns can be carried out by an offline mappings analysis identifying compatible templates; (2) if the ontology operation, the only possible unions between database columns are those induced by mappings with compatible templates; (3) the case of a non-empty ontology can always be reduced to the case of an empty ontology by exploiting the T-mapping technique. From (1)–(3), we conclude that it is possible to enumerate all possible SQL joins and unions by analyzing the T-mapping of an OBDF specification. These are used by Algorithm 1 to compute data hints for an OBDF instance. The number of hints is bound to \( n^{2} \), where \( n \) is the number of attributes in the DB schema. The worst case captures the scenario where every DB attribute is mapped to the same URI template. Algorithm 1 needs to be re-run every time the sources get updated, which makes our approach less suited to scenarios where sources are, e.g., streams of data.

5.2 Query Optimization Rules

We introduce a set of query optimization rules based on pre-computed hints. The top part of Figure 3 shows well-known rules commonly applied in OBDA systems [10, 35], which are relevant also in our approach. The bottom part introduces the novel rules based on hints, which we now explain in detail.

The empty join elimination rule ejr removes empty joins from SQL queries. To check the applicability of this rule, we rely on the observation that, if a join between two relations appears in the unfolding, then we have pre-computed it during the computation of hints. Therefore, given a set \( E \) of empty join hints, computed as in Algorithm 1, to verify \( A \bowtie B =_{\emptyset} \emptyset \) if it suffices to check whether this expression belongs to \( E \).

The containment elimination rule er removes redundant unions from SQL queries. As discussed for ejr, the applicability of this rule can be verified through a membership check over the set \( C \) of containment redundancies computed by Algorithm 1.

The equivalence elimination rule er replaces the operands of joins or left outer joins with expressions. The applicability of the rule involves checking a data containment (as for rule er), and a condition \( \dagger \) that is verified only if the cost of the resulting expression is less than the cost of the original one. In Section 5.3 we discuss a way to compute such cost.

Finally, the materialization rule mtr replaces the federated joins in the SQL query according to the views computed by Algorithm 1.

5.3 Cost Model

The application of the rules introduced in Section 5.2 is guided by a cost model, which associates an evaluation cost to each SQL query. Our cost model is based on the following heuristic arguments:

(1) Local joins are preferred to federated joins.
(2) Efficient sources should be favored over inefficient ones.
(3) Redundant and empty sub-expressions should be eliminated.
(4) Whenever available, materialized views are preferred.

Besides such heuristics specific to the federated setting, we also assume the standard heuristics for the OBDA setting [35]:

(5) URI templates should be applied at the highest level possible of the algebra tree.
(6) Joins should be pushed inside unions (see rule dir).
(7) Self-joins and redundant unions should be eliminated (see, e.g., rule sjr).

We now introduce our cost measure, inspired by these heuristic assumptions. For the remainder of this section, we fix a VDB instance \( D \) for \( \Sigma_{E} \).

A terminal federated join is a query of the form \( FJ = \left(\bigcup_{i=1}^{n} A_{i}\right) \circ \left(\bigcup_{j=1}^{m} B_{j}\right) \), where \( o \in \{\bowtie, \bowtie, \bowtie\} \), and each \( A_{i} \) and \( B_{j} \) is an expression over \( \Sigma_{E} \) not containing joins. The cost of evaluating \( FJ \) over \( D \) is estimated as:

$$\text{Cost}_{\text{base}}(FJ) = n + m.$$  

In compliance with heuristic (1) above, this measure favors the structure where unions are folded into a single federated join, as opposed to the UCQ structure (which would contain \( n + m \) possibly federated, joins). In compliance with (3) and (7), removing redundant operations from either operand will decrease the cost of the expression.

Consider a SQL query \( q \) over \( \Sigma_{E} \) containing \( k \) terminal federated joins \( FJ_{1}, \ldots, FJ_{k} \), where \( \text{Cost}_{\text{base}}(FJ_{i}) = c_{i} \) for \( 1 \leq i \leq k \), and let \( \#\text{ineff} = \sum S \in \{S | \text{src}(q) \mid S \text{ is inefficient} \} \). \text{occ}(S, q) \) denote the total

\footnote{Assuming the SPARQL 1.1 Entailment Regime and OWL 2 QL.}
We present a novel algorithm, whenever a materialized view is used, an inefficient or federated rule the algorithm iteratively applies rule $dlr$ between source relations. To perform this structural transformation, the algorithm attempts to remove joins. To this aim, a UCQ-like form maximizes the number of unions between source relations.

Note that our cost model is compliant with heuristics (1)-(7). For instance, if a structural transformation increases the number of joins in an expression, but these joins are local, then the cost of such expression will still decrease (see heuristics (5) and (6)). Also, whenever a materialized view is used, an inefficient or federated join is removed from the expression, reducing either argument of the cost value (heuristic (4)).

5.4 Hints-Based Unfolding Algorithm

We present a novel algorithm, $unfold_{OBDF}$, for translating federated SPARQL queries over an OBDF system into SQL queries over a VDF system. The algorithm relies on pre-computed hints, and applies the optimization rules discussed in Section 5.2, guided by the cost-model described in Section 5.3.

Algorithm 2 starts by computing a translation of a SPARQL query $Q$ w.r.t. $(T, M, \Sigma_\mathcal{E})$ (Line 1). For the translation it adopts $unfold_{wrap}$, discussed in Section 2, because such an algorithm maximizes the number of unions between source relations.

By hint construction, each redundant union must occur in $C$. Therefore, identifying redundant unions reduces to a set membership check (Lines 3 and 8). Line 4 uses the cost-model to determine the specific application branch of rule $er$, in case the containment holds in both directions.

After having removed as many redundant unions as possible, the algorithm attempts to remove joins. To this aim, a UCQ-like form is more suited, since it maximizes the number of join operations between source relations. To perform this structural transformation, the algorithm iteratively applies rule $dlr$ (Line 11).

Sub-procedure $applyExaustively$ (Line 12) removes redundant self-joins, and transforms federated joins or joins between inefficient sources either into local joins through the application of rule $er$, or into queries over the materialized views through the application of rule $mtr$. Observe that procedure $applyExaustively$ is terminating: each application of $ejr$, $sjr$, and $mtr$ reduces the size of $E$, and each application of $er$ reduces Cost(E). Again, to decide the applicability of each of these rules it suffices to solve a membership problem in $E$, $C$, or $M$.

Note that the application of rule $dlr$ in Line 11 potentially introduces a large number of joins, many of which might even be federated or over inefficient sources. Hence, even after the optimizations applied in Line 12, the algorithm tries to fold-back the structure of the query (Lines 13 and 14) through an inverse application of $dlr$.

After these steps, if the optimization of the federated join led to an improvement of the overall cost, then the changes are accepted and incorporated in $q$ (Lines 15 and 16).

After a similar strategy for left outer joins is applied, the algorithm returns the final translation, whose cost is lower than or equal to the cost of the initial unfolding.
Example 5.2. Consider again the OBDF specification and SPARQL query from Example 3.2, and an OBDF instance \((\mathcal{F}, \mathcal{D})\) of \(\mathcal{F}\). Suppose all "id" columns to be primary keys for the respective tables. Suppose we have the empty federated join hint \(\mathcal{G} \bowtie_{\text{id1}} \mathcal{G}_2 = \emptyset\) and the containment redundancy hint \(\pi_{\text{pid}/\text{pidname}}(\mathcal{PerInfo}_3) \bowtie \mathcal{D} \pi_{\text{sid}/\text{sidname}}(\mathcal{Employee}_4)\) on the basis of the empty join hint and the application of the rule \(\text{srj}\).
(1) Similarly, the intermediate \(q_1\) is translated into query \(q_2\).
(2) On the basis of containment redundancy hint and the given source labelling, the union between \(\mathcal{PerInfo}_3\) and \(\mathcal{Employee}_4\) is then removed by the application of rule \(\text{cr}\), and only the projection over the fastest source \(\mathcal{PerInfo}_3\) is kept in the resulting query \(q_3\).

Each unfold step reduces the query’s cost, and \(\text{Cost}(q_3) < \text{Cost}(q)\).

6 EVALUATION

We have carried out an extensive experiment to verify the effectiveness of the proposed optimizations. We ran the experiments on an HP Proliant server with 2 Intel Xeon X5690 Processors (each with 12 logical cores at 3.47 GHz), 106GB of RAM and five 1TB 15K RPM HDs. For database systems, we employed PostgreSQL 14, DB2 11.5.7.0, MySQL 8.0.28 and MS SQL Server 2019 as relational systems, MongoDB 4.4.13 as the NoSQL system, and Teiid 16.0.0 as the federation engine.

The material for reproducing the experiments and the appendix of this work are available at: https://github.com/efghk321456/sc.

6.1 Experiment Setup

Our experiment is based on the well-known Berlin SPARQL Benchmark (BSBM) [5]. BSBM is built around an e-commerce use case in which a set of products is offered by different vendors and consumers have posted reviews about products. The data is organized as ten relational tables.

Data sets. We generate 5 data sets out of the original BSBM tables, while introducing data partitioning and redundancy to simulate the scenarios where data from different sources are mapped to the same classes in the ontology. We split the tables Product, ProductFeatureProduct, and ProductTypeProduct "horizontally" according to their product IDs, into two data sets \(D_1\) and \(D_2\). We make a copy of the table Review and put it into \(D_1\). Dataset \(D_2\) contains ProductType and ProductFeature; \(D_3\) contains Offer, Producer, and Vendor; \(D_4\) contains Review and Person.

Data sources. We store the 5 data sets in different database systems and derive 8 data sources in total: \(D_1 - D_3\) are stored in RDBs and data sources \(S_1 - S_5\) are obtained. We convert the tables in \(D_2\) and \(D_3\) to CSV files to obtain two more data sources \(S_6\) and \(S_7\). We convert \(D_3\) into JSON files, stored them in MongoDB, and obtain a data source \(S_8\).

OBDA/OBDF specifications. We use the ontology and mappings from [44] for OBDA, with minor modifications on the mappings for handling the different DBs. As base lines, we generate two OBDA specifications using two centralized RDBs: sc1, containing the original BSBM tables, and sc2, containing the tables in \(S_1 - S_5\). We create two OBDF specifications over Teiid: a homogeneous (relational) one, Hom, defined over sources \(S_1 - S_5\), and a heterogeneous one (in which some data are also in CSV files and MongoDB) het, defined over the sources \(S_1, S_2, S_3, S_4, S_5\). For each OBDF specification, the hints include 3 empty federated joins, 1 data redundancy, and 6 materialized views (see the appendix). The materialized views are stored in a local PostgreSQL DB.

6.2 Query Evaluation and Result Analysis

Table 1: Performance change compared with the sc1 setting

<table>
<thead>
<tr>
<th></th>
<th>sc1</th>
<th>sc2</th>
<th>hom</th>
<th>homopt</th>
<th>homopt</th>
<th>het</th>
<th>hetopt</th>
<th>hetopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>200K</td>
<td>1.0</td>
<td>8.4</td>
<td>8.5</td>
<td>0.6</td>
<td>0.4</td>
<td>254.9</td>
<td>54.6</td>
<td>12.2</td>
</tr>
<tr>
<td>2M</td>
<td>1.0</td>
<td>59.7</td>
<td>67.5</td>
<td>1.8</td>
<td>0.6</td>
<td>922.9</td>
<td>205.7</td>
<td>52.1</td>
</tr>
</tbody>
</table>

For scalability testing, we generate three groups of instances using the BSBM data generating tool, setting the numbers of products to be 20K, 200K and 2M. In this way each OBDA/OBDF specification has 3 instances. For each OBDF instance, the hints, including data redundancy, empty federated joins, and materialized views, are pre-computed following the approach in Section 5.1. For space reasons, we only report the results on 200K and 2M. Further details are listed in the Appendix.

We consider the 12 SPARQL queries \(Q_1\) to \(Q_{12}\) from the BSBM benchmark. The SQL queries without optimization with hints are generated by Ontop, and the optimized ones are computed manually following the algorithms in Section 5. All queries were ran 12 times:
2 warm-up runs, and 10 test runs. In each run, the queries were instantiated with different parameters. The tests were run by using the testing platform of the NPD benchmark [26].

The SQL-queries evaluation times are reported in Figure 5. In such figure, homopt and hetopt denote the evaluation with the hints of empty federated joins and redundancies; and hom\textsuperscript{matv} and het\textsuperscript{matv} employed all the hints including materialized views.

To highlight the differences between the setups, we compute the aggregated ratios of running times compared with sc1, reported in Table 1. Concretely, for each setup c, the values are computed by \( \left( \prod_{i=1}^{N} \frac{t_i^{c}}{t_i^{sc1}} \right)^{\frac{1}{N}} \), where \( N=12 \) is the number of queries, and \( t_i^{c} \) and \( t_i^{sc1} \) are the times of evaluating \( Q_i \) in setup \( c \) and sc1, respectively.

Table 2 reports the hints used for each query, and the data sources accessed before and after the optimization with respect to the hints. Hereafter, we summarize the main outcomes of our experiments:

<table>
<thead>
<tr>
<th>Query</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hints</td>
<td>EFJ</td>
<td>EFJ</td>
<td>EFJ</td>
<td>EFJ</td>
<td>EFJ</td>
<td>DR</td>
<td>EFJ</td>
<td>DR</td>
<td>MatV</td>
<td>MatV</td>
<td>MatV</td>
<td>MatV</td>
</tr>
<tr>
<td>Before</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td></td>
</tr>
<tr>
<td>After</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td>S1, S2</td>
<td></td>
</tr>
</tbody>
</table>

7 RELATED WORK

OBDA [12, 33] is a semantic technology-based paradigm that has been developed since the mid 2000s [42, 43] with the aim to ease the access to legacy data stored in relational data source. Most research in this field has focused on query rewriting [8, 9, 13, 18, 24, 25, 32, 41], that is, on the problem of reformulating queries over the virtual RDF graph into an equivalent query over the data source.

Data federation studies the problem of accessing multiple, distributed and possibly heterogeneous data sources [2, 3, 16, 17, 23, 29, 31, 36, 37, 39, 40] via a unified schema. Many mature and high-optimized data federation systems have been developed in both academia and industry (e.g., ANAPSID, Teiid, and Denodo). For a comprehensive and up-to-date survey of these systems, and a brief overview on the main optimization techniques used therein, the interested reader might refer to [19].

8 CONCLUSION

This work introduces the Ontology-based Data Federation setting and studies the problem of optimizing query translations in this setting. We provide techniques to address this problem, which are based on source data information that can be automatically computed in an offline stage by exploiting the information encoded in an OBDF specification. We performed an extensive empirical evaluation, showing that our techniques have a significant impact on the overall performance of query answering.

In this work we laid the foundations of OBDF. In future work we plan on further investigating hint-based optimizations, as well as implementing our algorithms in an actual system and applying OBDF to tackle complex, real-world scenarios. A more sophisticated handling of static and dynamic sources (e.g., [6]) might be necessary in these scenarios.

ACKNOWLEDGMENTS

This research has been partially supported by the EU H2020 project INODE (grant agreement No. 863410), by the Italian PRIN project HOPE (2019-2022), by the Free University of Bozen-Bolzano through the project MP4OBDA, and by the “Fusion Grant” project HIVE.


