

Ultrawrap: SPARQL Execution on Relational Data

Technical Report

Juan F. Sequeda and Daniel P. Miranker
Department of Computer Science
University of Texas at Austin
{jsequeda, miranker} @ cs.utexas.edu

Abstract: The Semantic Web’s promise to achieve web-wide data integration requires the inclusion of legacy relational data as RDF, which, in turn, requires the execution of SPARQL queries on the legacy relational database. In this paper we explore a hypothesis: *existing commercial relational databases already subsume the algorithms and optimizations needed to support effective SPARQL execution on existing relationally stored data.* The experiment, embodied in a system called *Ultrawrap*, comprises encoding a logical representation of the database as a graph using SQL views and a simple syntactic translation of SPARQL queries to SQL queries on those views. Thus, in the course executing a SPARQL query, the SQL optimizer both instantiates a mapping of relational data to RDF and optimizes its execution. Other approaches typically implement aspects of query optimization and execution outside the SQL environment.

Ultrawrap is evaluated using two benchmarks across the three major relational database management systems. We identify two important optimizations: detection of unsatisfiable conditions and self-join elimination, such that, when applied, SPARQL queries execute at nearly the same speed as semantically equivalent native SQL queries, providing strong evidence of the validity of the hypothesis.

1. INTRODUCTION

The thesis of this paper is that by carefully constructing SQL views to create a representation of a legacy relational database as an RDF graph [6], the existing algorithmic machinery in SQL optimizers is already sufficient to effectively execute SPARQL queries on native relational data. A complete system, Ultrawrap, is built. Ultrawrap is organized as a set of four compilers, where the SQL optimizer acts as one of the compilers. The result is a system that is simple and robust, as all intricate aspects of the implementation have been pushed down to established SQL query systems. Further, the approach enables real-time consistency between the relational and the RDF representations of the data.

To be clear, this paper concerns RDF wrappers for relational database management systems (RDBMSs), also known as *Relational Database to RDF* systems (RDB2RDF). These systems are distinguished from triplestores, which are DBMSs for RDF data. Some triplestores use RDBMSs as their backend. The intention of RDB2RDF systems is to bridge the Semantic Web with legacy data stored in relational databases. Approximately 70% of websites are backed up by relational databases [32]; therefore the success of the Semantic Web depends on mapping this legacy relational data to RDF. Standardization for such mapping is the subject of a W3C working group [16, 29].

In 2008, Angles and Gutierrez showed that SPARQL is equivalent in expressive power to relational algebra [15]. Thus, one might expect the validity of the thesis to be a foregone conclusion. However, in 2009, two studies that evaluated three systems, D2R, Virtuoso RDF Views and Squirrel RDF, came to the opposite conclusion; existing SPARQL to SQL translation systems do not compete with traditional relational databases [20, 31].

A motivation for Ultrawrap is to resolve the apparent contradiction among the aforementioned papers. Our supposition is that the SQL query execution speed of enterprise class RDBMS, is excellent, and indicative of the fastest executions possible. Thus, we sought to organize an RDB2RDF architecture that maximized the use of existing SQL query systems to optimize the query. We identified two SPARQL benchmark suites where the data originates in a relational schema and the benchmark provides semantically equivalent queries in both SPARQL and SQL, and used both representations to evaluate the system.

To maximize the use of existing SQL query systems, Ultrawrap itself is minimal. Since query optimization integrates both logical and physical transformations, even the transformation of a SPARQL query plan on a graph representation of data, to a SQL query plan on relational data is implemented by an existing SQL query system. This is accomplished by encoding the mapping of the representation of relational data from tuples to graphs as an *unmaterialized* view. In Semantic Web practice, RDF graphs are represented as sets of directed labeled edges, a.k.a. *triples*. We call the mapping from tuples to graphs, the *Tripleview*. The core definition of such a mapping is shared among many results [42] and is now the subject of an emerging W3C standard [16].

Our initial approach was to represent the mapping in a three-column Tripleview. However, very quickly we realized that given this structure, the optimizer would not leverage indexes. The Tripleview was refined to reflect physical schema elements including indexes and datatypes. Nevertheless, the Tripleview is a logical representation of the relational data as triples.

In keeping with minimizing the implementation, the base implementation of Ultrawrap limits the translation of SPARQL queries to SQL queries (on the Tripleview) to simple string substitutions of SPARQL syntax with SQL syntax. This is straightforward as not only is the expressive power of SPARQL subsumed by SQL, the semantics of

many SPARQL query operators are defined in the SPARQL language specification by asserting their equivalence to relational operators [37].

We find that using Ultrawrap and two RDB2RDF benchmarks, the Berlin SPARQL Benchmark (BSBM) [1] and Barton [14], that the execution time of most queries on all three enterprise class RDBMSs is comparable to the execution time of semantically equivalent native SQL queries (hereafter native SQL). By native SQL we mean SQL queries that have been written by users having knowledge of the underlying relational schema.

We find that two logical transformations appear to be critical to achieving SPARQL query plans identical to plans generated for native SQL queries: (1) detection of unsatisfiable conditions and (2) self-join elimination. We observe that SQL Server only implements the first optimization; Oracle only implements the second optimization, DB2 implements both.

The detection of an unsatisfiable conditions optimization was implemented in an augmented version of Ultrawrap. The net performance enables us to assess the relative importance of the two optimization and helps isolate and identify additional concerns. Removal of self-joins is the less important of the two transformations. No RDBMS removes self left-outer joins.

The single most challenging queries are those where the SPARQL query has a predicate variable that maps to an attribute name. Such queries expand to querying all tables while a native SQL query may be limited to a few tables or even just one query. Otherwise, all benchmark queries ran in comparable time to the native SQL, except when the optimizer failed to the best join order.

Contributions: The main contributions of this paper are:

1. Ultrawrap, an RDB2RDF system with a novel architecture to representing relational data as RDF triples using unmaterialized views and which can answer SPARQL queries by translating them to SQL queries over the views (Section 3).
2. Identification of two important optimizations that support efficient SPARQL execution: detection of unsatisfiable conditions and self-join elimination (Section 4).
3. A comprehensive performance evaluation, Berlin SPARQL Benchmark with 100 million triples and Barton with 50 million triples, on three major RDBMS: IBM DB2, Microsoft SQL Server and Oracle demonstrating that Ultrawrap is able to execute most SPARQL queries at comparable execution time as the native SQL queries (Section 5).

2. RELATED WORK

RDF and SPARQL are the technological building blocks of the Semantic Web vision; a web-wide integration of data [19]. These languages have spawned research into general purpose DBMS founded on the RDF graph model, *triplestores*, implemented as either native RDF triplestores [36, 47] or RDF triplestores backed by SQL databases [13, 22, 28, 48].

Ultrawrap is concerned with wrapping relational databases and using the existing SQL system to execute SPARQL queries on the relationally stored data without replicating the data to triplestores. Systems with similar function to Ultrawrap are: D2R, ODEMapster/R2O, Spyder, SquirrelRDF and Virtuoso RDF Views [2, 3, 18, 8, 9, 10]. It is commonly understood that these systems implement query rewrite optimizations. Despite the fact that none of these systems have published a refereed scientific paper describing their rewriting algorithms and optimizations, open-source code and forums¹²³ provide evidence of their architecture. For example, we observed that for some SPARQL queries D2R generates multiple SQL queries and necessarily executed a join among those results outside of the database.

Chebotko et. al. presents a semantics preserving translation of SPARQL to SQL, which considers any generic underlying SQL schema, i.e Triple table schema to represent RDF data, etc. However, they only evaluate their translation on triple-table schemas with 1 million triples. The generated SQL resembles the relational algebra rules used to define the semantics SPARQL, resulting in “multiple coalesce functions in one projection, null-accepting predicates, and outer union implementations” [26]. Elliot et al builds on Chebotko et al. by presenting algorithms that generate a flat SQL query. This algorithm is also generic with respect to the underlying SQL schema. Their evaluation was also on a triple-table style schema on small datasets of 2-5 million triples [30]. While in theory, these two SPARQL to SQL translations approaches should function as a RDB2RDF system on any SQL database, both their evaluations were limited to building a triplestore on top of a relational database.

While studies that evaluate triplestores abound, we have only been able to identify the two studies from 2009 that compare SPARQL execution of RDB2RDF systems with native SQL execution on the relational database [20, 31]. Bizer & Schultz compared D2R and Virtuoso RDF Views on MySQL [20]. Gray et al compared D2R and SquirrelRDF on MySQL [31].

The March 2009 Berlin SPARQL Benchmark on the 100 million triple dataset reported that SPARQL queries on the evaluated RDB2RDF systems were up to 1000 times slower than the native SQL queries! Today, those systems are

¹http://sourceforge.net/mailarchive/message.php?msg_id=27731620

²http://sourceforge.net/mailarchive/message.php?msg_id=28142066

³http://sourceforge.net/mailarchive/message.php?msg_id=28051074

still the most used in the Semantic Web community and no new system has been introduced and evaluated since then. Bizer & Schultz [20], creators of the Berlin SPARQL Benchmark, concluded that: “Setting the results of the RDF stores and the SPARQL-to-SQL rewriters in relation to the performance of classical RDBMS unveiled an unedifying picture. Comparing the overall performance (100M triple, single client, all queries) of the fastest rewriter with the fastest relational database shows an overhead for query rewriting of 106%. This is an indicator that there is still room for improving the rewriting algorithms.”

Gray et al [31] tested D2R and SquirrelRDF on a scientific database. This study concluded that “... current rdb2rdf systems are not capable of providing the query execution performance required to implement a scientific data integration system based on the rdf model. [...] it is likely that with more work on query translation, suitable mechanisms for translating queries could be developed. These mechanisms should focus on exploiting the underlying database system’s capabilities to optimize queries and process large quantities of structure data, e.g. pushing the selection conditions to the underlying database system.”

Other RDB2RDF systems go through a cumbersome ETL process of extracting relational data, translating it to RDF and loading the results into a triplestore [38]. In this case, two copies of the same data must be maintained.

Related studies have compared native triplestores with RDB2RDF systems and native triplestores with relational database. In 2007, Svihla & Jelinek determined that RDB2RDF systems are faster than the Jena and Sesame triplestores [45]. In 2009, Schmidt et al compared Sesame triplestore with the triple table, vertical partitioned storage scheme and the native relational scheme on MonetDB, a column-store relational database. This study concluded that none of the RDF schemes was competitive to the native relational scheme [39]. In 2010, MahmoudiNasab and Sakr also compared the triple table, property table and vertical partitioned storage scheme with the native relational scheme on IBM DB2. They also concluded that none of the storage schemes compete with the native relational scheme [35]. In conclusion, native SQL queries on relationally stored data outperform any other approach.

Ontology-based data access systems, such as Quest, MASTRO and ONDA [5, 23, 24] focus on mapping relational databases to ontologies in order to perform reasoning during query execution. These systems are restricted to conjunctive queries and do not fully support SPARQL.

3. ULTRAWRAP

Ultrawrap [41] is motivated to test our hypothesis, that existing commercial relational databases already subsume the algorithms and optimizations needed to support effective SPARQL execution on existing relationally stored data. Ultrawrap supports both the W3C’s Direct Mapping [16] and R2RML [29]. An additional distinguishing feature of Ultrawrap is its support for an augmented direct mapping that also translates the relational schema into an ontology and has been proven to be information and query preserving [40]. The starting point of Ultrawrap is the augmented direct mapping, hence, a user does not have to have knowledge of the relational schema, learn a mapping language or manually create the mapping. Nevertheless, a customized mapping can be presented in R2RML.

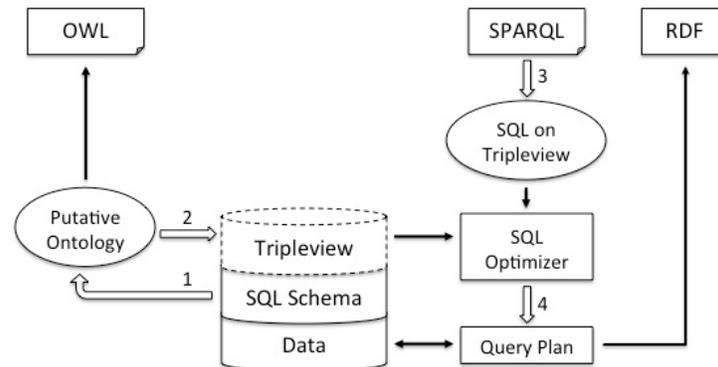


Figure 1. Architecture of Ultrawrap

Ultrawrap is comprised of four primary components as shown in Figure 1:

1. The translation of a SQL schema, including constraints, to an OWL ontology: the putative ontology (PO) [40, 46].
2. The creation of an intentional triple table in the database by augmenting the relational schema with one or more SQL Views: the Tripleview.
3. Translation of SPARQL queries to equivalent SQL queries operating on the Tripleview.
4. The native SQL query optimizer, which becomes responsible for rewriting triple based queries and effecting their execution on extensional relational data.

These four components can be seen as four different language compilers. As an ensemble, the first three provide for the logical mapping of schema, data and queries between the relational and Semantic Web languages. The fourth component, the SQL optimizer, is responsible for all the physical mappings and their implementation.

3.1 Compilation

The components of Ultrawrap may also be decomposed as a compilation phase and a runtime phase. The goal of the compilation phase is the creation of the Tripleview. The first step in the compilation phase is the translation of the application's SQL schema to OWL.

3.1.1 Generating the Putative Ontology

An RDF triple consists of a subject, predicate and object. The subject and object form graph nodes. The predicate is the labeled edge. The schema for RDF can be represented in two different Semantic Web languages: RDF-Schema [7] or OWL (Web Ontological Language) [11]. For example, in the RDF triple <Product123, hasFeature, ProductFeature456>, the predicate *hasFeature* has a domain of class Product and a range of class ProductFeature. We can then infer that Product123 is of type Product and ProductFeature456 is of type ProductFeature.

Relational data is to RDF as relational schema is to RDF-Schema or OWL; hence in order to translate relational data to RDF, the first step is to identify the ontological representation of the relational schema. We implement the transformation rules for SQL-DDL to OWL of Sequeda et al [40], which considers the SQL-DDL and the integrity constraints (foreign keys and primary keys). Briefly, this transformation consists of representing tables as ontological classes, foreign key attributes of a table as object properties and all other attributes as datatype properties. Tables that represent a many-to-many relationship are translated to object properties. Each property has its respective domain and range. Both datatype and object properties have classes as domains. Datatype properties have a datatype as a range while object properties have a class as its range. An emerging standard stipulates many of these same mappings [16].

When a SQL schema encodes a sophisticated logical data model, the OWL ontology produced by Ultrawrap can accurately be called an ontology. Since the quality of a SQL application's data modeling effort is highly variable, and the meaning of a *good ontology* is subjective, we allay controversy by calling the result collection of OWL declarations a putative ontology (PO).

3.1.2 Creating the Tripleview

The putative ontology provides the logical definition of the RDF representation of relational data. The second step of the compilation phase is to generate the Tripleview, an intentional representation of the relational data as RDF expressed with SQL views. The Tripleview is a SQL view comprised of the union of SELECT-FROM-WHERE (SFW) statements that returns triples consistent with the putative ontology. Given that null values are not expressible in RDF, the WHERE clause filters attributes with null values (IS NOT NULL). Instead of using URIs to uniquely identify the subject, predicate and objects, we use a concatenation of either the table name with the primary key value or table name with attribute name.

Due to its simplicity, our starting point is a representation commonly known as the *triple table* approach; that is a table with three attributes: subject, predicate and object. Many triplestores that use RDBMSs as their back-ends use a triple table [28, 48]. However, studies have shown that storing RDF with the triple table approach in a relational database is easily improved upon [13, 35]. This issue is not directly relevant to Ultrawrap because the relational data is not being materialized in a triple table.

Even though our goal is to define a virtual triple model, we still have to anticipate the physical characteristics of the database and the capacity of the SQL optimizer to produce optimal physical plans. Toward that end, we have identified two refinements to the Tripleview.

Refinement 1: Our initial approach was to create a single Tripleview with 3 attributes: <subject, predicate, object>. The subject corresponds to concatenating the name of the table and the primary key value. The predicate is a constant value that corresponds to each attribute name of each table. There can be two types of object values: a value from the database or a concatenation of the name of a table with its primary key value. However, joins were slow because the optimizer was not exploiting the indexes on the primary keys. Therefore, the Tripleview was extended to consist of 5 attributes: <subject, primary key of subject, predicate, object, primary key of object>. Separating the primary key in the Tripleview allows the query optimizer to exploit them because the joins are done on these values. If the object is a value, then a NULL is used as the primary key of the object. The subject and object are still kept as the concatenation of the table name with the primary key value because this is used to generate the final URI, which uniquely identifies each tuple in the database. For simplicity, composite keys were not considered in the Tripleview. Nevertheless, it is possible to augment the amount of attributes in the Tripleview to include each separate key value.

Refinement 2: Refinement 1 represented the entire database in a single Tripleview. This meant that all values were cast to the same datatype (namely varchar). However, the size of the object field of the Tripleview is the size of the largest varchar which led to poor query performance. Due to this issue, it was beneficial to create a separate Tripleview for each datatype. For varchar, this includes each length declared in the schema. For example, datatypes with

varchar(50) and varchar(200) are considered different. Using multiple Tripleviews requires less bookkeeping than one might anticipate. Each attribute is mapped to its corresponding Tripleview and stored in a hashtable. Then, given an attribute, the corresponding Tripleview can be retrieved immediately.

Table 1 shows an example relational database and the logical contents of the Tripleviews are shown in Table 2-5. Pseudo-code for creating the Tripleviews is shown in Figures 2-5. Figure 6 shows the CREATE VIEW statements for the Tripleviews.

Table 1. Example of Product and Producer table

Product					Producer		
Id	Label	pNum1	pNum2	prodFK	Id	title	location
1	ABC	1	2	4	4	Foo	TX
2	XYZ	3	3	5	5	Bar	CA

Table 2. Logical contents of Tripleview for types

S	S_ID	P	O	O_ID
Product1	1	type	Product	NULL
Product2	2	type	Product	NULL
Producer4	4	type	Producer	NULL
Producer5	5	type	Producer	NULL

Table 3. Logical contents of Tripleview for varchar(50)

S	S_ID	P	O	O_ID
Product1	1	Product#label	ABC	NULL
Product2	2	Product#label	XYZ	NULL
Producer4	4	Producer#title	Foo	NULL
Producer4	4	Producer#location	TX	NULL
Producer5	5	Producer#title	Bar	NULL
Producer5	5	Producer#location	CA	NULL

Table 4. Logical contents of Tripleview for int

S	S_ID	P	O	O_ID
Product1	1	Product#pNum1	1	NULL
Product1	1	Product#pNum2	2	NULL
Product2	2	Product#pNum1	3	NULL
Product2	2	Product#pNum2	3	NULL

Table 5. Logical contents of Tripleview for object properties

S	S_ID	P	O	O_ID
Product1	1	Product#producer	Producer4	4
Product2	2	Product#producer	Producer5	5

```

PO ← Transform SQL-DDL to OWL
list ← empty list
for each ontological object x of PO
  if x is a OWL Class then
    pk = getPrimaryKey(x)
    S ← SELECT concat(x,pk) as s, pk as s_id, 'type' as p, x as o, null as o_id FROM x
    add S to list
  end for each
return createTripleview('Tripleview_type', list)

```

Figure 2. Pseudo-code to create a Tripleview for types

```

PO ← Transform SQL-DDL to OWL
list ← empty list
for each ontological object x of PO
  if x is a OWL Datatype Property then
    datatype = getDatatype(x)
    if datatype == varchar then
      domain = getDomain(x)
      pk = getPrimaryKey(domain)
      S ← SELECT concat(domain,pk) as s, pk as s_id, 'x' as p, x as o, null as o_id FROM domain
      add S to list
  end for each
return createTripleview("Tripleview_varchar", list)

```

Figure 3. Pseudo-code to create a Tripleview for Varchar Datatype Properties

```

PO ← Transform SQL-DDL to OWL
list ← empty list
for each ontological object x of PO
  if x is a OWL Object Property then
    domain = getDomain(x)
    d_pk = getPrimaryKey(domain)
    range = getRange(x)
    r_pk = getPrimaryKey(range)
    S ← SELECT concat(domain, d_pk) as s, d_pk as s_id, concat(domain,range) as p, concat(range, r_pk) as o, r_pk as o_id
    FROM x
    add S to list
  end for each
return createTripleview("Tripleview_object", list)

```

Figure 4. Pseudo-code to create a Tripleview for Object Properties

```

TripleView ← 'CREATE VIEW name (s, s_id, p, o, o_id) AS'
for each element l in list
  if l is last element in list then
    TripleView ← l
  else
    TripleView ← l + 'UNION ALL'
  end for each
return TripleView

```

Figure 5. Pseudo-code to create the createTripleview method

```

CREATE VIEW Tripleview_type(s, s_id, p, o, o_id) AS
SELECT "Product"+id as s, id as s_id, "type" as p, "Product" as o, null as o_id FROM Product
UNION ALL
SELECT "Producer"+id as s, id as s_id, "type" as p, "Producer" as o, null as o_id FROM Producer;

CREATE VIEW Tripleview_varchar50(s, s_id, p, o, o_id) AS
SELECT "Product"+id as s, id as s_id, "label" as p, label as o, null as o_id FROM Product WHERE label IS NOT NULL
UNION ALL
SELECT "Producer"+id as s, id as s_id, "title" as p, title as o, null as o_id FROM Producer WHERE title IS NOT NULL
UNION ALL
SELECT "Producer"+id as s, id as s_id, "location" as p, location as o, null as o_id FROM Producer WHERE location IS NOT NULL;

CREATE VIEW Tripleview_int(s, s_id, p, o, o_id) AS
SELECT "Product"+id as s, id as s_id, "pNum1" as p, pNum1 as o, null as o_id FROM Product WHERE pNum1 IS NOT NULL

```

```

UNION ALL
SELECT "Product"+id as s, id as s_id, "pNum2" as p, pNum2 as o, null as o_id FROM Product WHERE pNum2 IS NOT NULL;

CREATE VIEW Tripleview_object(s, s_id, p, o, o_id) AS
SELECT "Product" id as s, id as s_id, "product_producer" as p, "Producer"+prodFk as o, prodFk as o_id FROM Product

```

Figure 6. CREATE statements defining the Tripleviews

3.2 RUNTIME

Ultrawrap’s runtime phase encompasses the translation of SPARQL queries to SQL queries on the Tripleviews and the maximal use of the SQL infrastructure to do the SPARQL query rewriting and execution.

3.2.1 SPARQL to SQL translation

SPARQL is a graph pattern matching query language [37] that has the form:

```

SELECT ?var1 ?var2 ...
WHERE{ triple-pattern-1.
      triple-pattern-2.
      ... }

```

where each triple-pattern consists of a subject, predicate, object and any of these can be a variable or a constant. Variables can occur in multiple triple-patterns implying a join. Consider the following SPARQL query as our running example:

```

SELECT ?label ?pnum1
WHERE{ ?x label ?label.
      ?x pnum1 ?pnum1.}

```

This SPARQL query binds the predicate of the first triple pattern to the constant “label” and the predicate of the second triple-pattern to the constant “pnum1”. The variable “?x” indicates that the results of triple-pattern-1 and triple-pattern-2 are to be joined and the final result is the projection of the binding to the variable “?label” and “?pnum1”.

The translation of the SPARQL query to a SQL query on the Tripleviews follows a classic compiler structure: a parser converts the SPARQL query string to an Abstract Syntax Tree (AST). The AST is translated into an SPARQL algebra expression tree. The SQL translation is accomplished by traversing the expression tree and replacing each SPARQL operator. Each internal node of the expression tree represents a SPARQL binary algebra operator while the leaves represent a Basic Graph Patterns (BGP), which is a set of triple patterns. A SPARQL BGP is a set of triple patterns where each one maps to a Tripleview. A SPARQL Join maps to a SQL Inner Join, a SPARQL Union maps to the SQL Union, a SPARQL Optional maps to SQL Left-Outer Join. In the previous example, there is only one BGP with two triple patterns and a Join between the triple patterns. The resulting SQL query is:

```

SELECT t1.o AS label, t2.o AS pnum1
FROM tripleview_varchar50 t1, tripleview_int t2
WHERE t1.p = 'label' AND t2.p = 'pnum1' AND t1.s_id = t2.s_id

```

Hereafter, this is called the Ultrawrap query. Note that the mapping mentioned in Refinement 2 (Section 2.1.2) was used in order to know which Tripleview to use. At the initial setup of the runtime, a hash table with the contents of the mapping is created. Therefore given an attribute such as label (key), the mapped Tripleview, in this case tripleview_varchar50 (value) can be retrieved immediately.

3.2.2 SQL engine is the Query Rewriter

Given the Ultrawrap query to be executed on the Tripleviews, the query is executed and it is observed how the SQL engine operates. These results are described in the following section. A main concern is if the SQL query can actually be parsed and executed on the Tripleviews, given the view is a very large union of a large amount of SFW statements. In the evaluation, BSBM consisted of 10 tables with a total of 78 attributes and Barton consisted of 20 tables with a total of 61 attributes. It is our understanding that SQL Server has a limit of 256 tables in a query [4]. Having Tripleviews for specific datatypes and their corresponding lengths can overcome this limit.

4 TWO IMPORTANT OPTIMIZATIONS

Upon succeeding in *ultrawrapping* different RDBMSs and reviewing query plans, two relational optimizations emerged as important for effective execution of SPARQL queries: 1) detection of unsatisfiable conditions and 2) self-join elimination. Perhaps, not by coincidence, these two optimizations are among *semantic query optimization (SQO)*

methods introduced in the 1980's [25, 27, 43]. In SQO, the objective is to leverage the semantics, represented in integrity constraints, for query optimization. The basic idea is to use integrity constraints to rewrite a query into a semantically equivalent one. These techniques were initially designed for deductive databases and then integrated in commercial relational databases [27].

Figure 7 shows the logical query plan of the Ultrawrap SQL query from the running example. This section describes how the query plan evolves through these optimizations. Describing a general-purpose implementation of these optimizations is not in the scope of this paper and we refer the reader to [25, 27] for more detail. In this query plan, for each of the triple patterns in the query, the Tripleview is accessed, which is consequently a union of all the SFW statements.

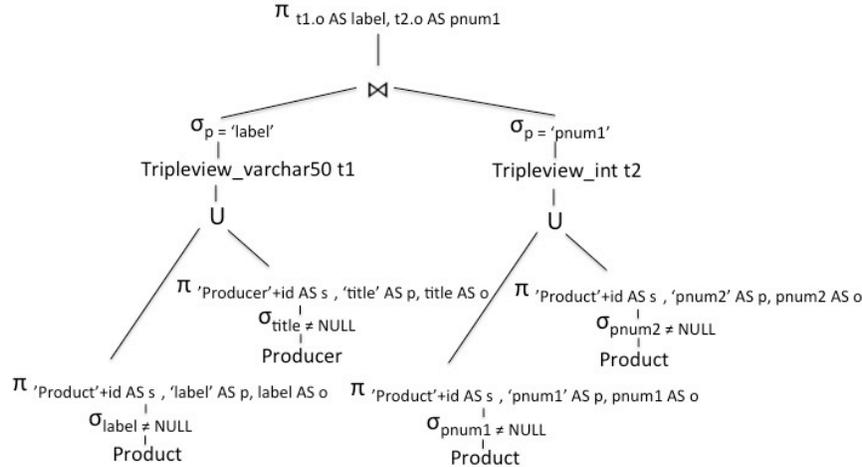


Figure 7. Initial query plan of the running example

4.1 Detection of Unsatisfiable Conditions

The idea of this optimization is to determine that the query result is empty if the existence of another answer would violate an existing condition; hence, the database does not need to be accessed [25]. Ultrawrap benefits from the following transformations that determine unsatisfiable conditions:

Elimination by Contradiction: Consider a query `SELECT * FROM R WHERE A = x AND A = y` such that $x \neq y$. Then the result of that query is empty. For example, it is clear that the query `SELECT * FROM Product WHERE ProductID = 1 AND ProductID = 2` will never return results.

Unnecessary Union Sub-tree Pruning: Given a query that includes the UNION operator and where it has been determined that an argument of the UNION is empty; then the corresponding argument can be eliminated. For example:

$$\begin{aligned} \text{UNION ALL } (\{\}, S, T) &= \text{UNION ALL } (S, T) \\ \text{UNION ALL } (\{\}, T) &= T \end{aligned}$$

In Ultrawrap's Tripleview, the constant value in the predicate position acts as the integrity constraint. Consider the following Tripleview:

```
CREATE VIEW Tripleview_varchar50(s,s_id,p,o,o_id) AS
SELECT 'Person'+id as s, id as s_id, 'name' as p, name as o, null as o_id FROM Person WHERE name IS NOT NULL
UNION ALL
SELECT 'Product'+id as s, id as s_id, 'label' as p, label as o, null as o_id FROM Product WHERE label IS NOT NULL
```

Now consider the following query "return all labels":

```
SELECT o FROM TripleView_varchar50 WHERE p = 'label'
```

The first SFW statement from Tripleview_varchar50 defines $p = \text{'name'}$ for every single query while the query contains $p = \text{'label'}$. With this contradiction, this particular SFW statement of Tripleview_varchar50 can be substituted by the empty set.

Since the Tripleview's definition includes all possible columns, any specific SPARQL query will only need a subset of the statements defined in the view. Once the elimination by contradiction transformation happens, all the unnecessary UNION ALL conditions are removed. When these two transformations are combined, the Tripleview is reduced to the specific subset of referenced columns.

With this optimization, the query plan in Figure 7 is optimized to have the exact SFW statements that are needed. See Figure 8.

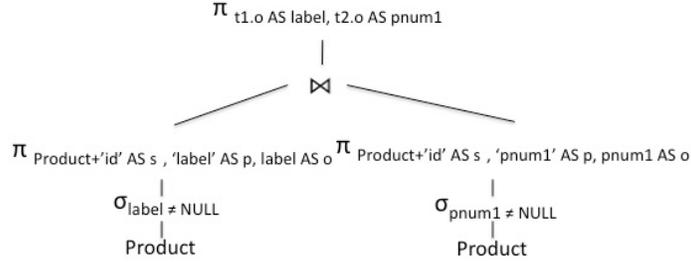


Figure 8. Query plan after application of Detection of Unsatisfiable Conditions optimization

4.2 Augmenting Ultrawrap

The Ultrawrap architecture is readily extended to include detection of unsatisfiable conditions optimization. By creating such a version, *Augmented Ultrawrap*, we are able to conduct a controlled experiment (see Section 5.3.2). Instead of creating a mapping between each attribute in the database and its corresponding Tripleview, a mapping is created for each attribute to its corresponding SFW statement. For example, attribute label is mapped to the SQL query: `SELECT 'Product'+id as s, id as s_id, 'label' as p, label as o, null as o_id FROM Product WHERE label IS NOT NULL`. At the initial setup of the runtime, a hash table with the contents of this mapping is generated. Therefore given an attribute such as label (key), the mapped SFW statement (value) can be retrieved immediately. Now, the view definition nested in the SQL query's FROM clause is replaced with the SFW statement.

4.3 Self-join Elimination

Join elimination is one of the several SQO techniques, where integrity constraints are used to eliminate a literal clause in the query. This implies that a join could also be eliminated if the table that is being dropped does not contribute any attributes in the results [25]. The type of join elimination that is desired is the *self-join elimination*, where a join occurs between the same tables. Two different cases are observed: *self-join elimination of projection* and *self-join elimination of selections*.

Self-join elimination of projection: This occurs when attributes from the same table are projected individually and then joined together. For example, the following unoptimized query projects the attributes label and pnum1 from the table product where id = 1, however each attribute projection is done separately and then joined:

```
SELECT p1.label, p2.pnum1 FROM product p1, product p2 WHERE p1.id = 1 and p1.id = p2.id
```

Given a self-join elimination optimization, the previous query may be rewritten as:

```
SELECT label, pnum1 FROM product WHERE id = 1
```

Self-join elimination of selection: This occurs when a selection on attributes from the same table are done individually and then joined together. For example, the following unoptimized query selects on pnum1 > 100 and pnum2 < 500 separately and then joined:

```
SELECT p1.id FROM product p1, product p2 WHERE p1.pnum1 >100 and p2.pnum2 < 500 and p1.id = p2.id
```

Given a self-join elimination optimization, the previous query may be rewritten as:

```
SELECT id FROM product WHERE pnum1 > 100 and pnum2 < 500
```

Figure 9 shows the final query plan after the self-joins are removed.

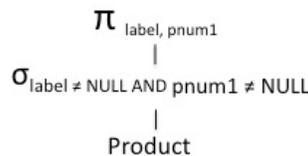


Figure 9. Query plan after self-join elimination optimization

5 EVALUATION

The evaluation requires workloads where the SPARQL queries anticipated that the RDF data was derived from a relational database. Two existing benchmarks fulfill this requirement. The Berlin SPARQL Benchmark (BSBM) [1] which is intended to imitate the query load of an e-commerce website and the Barton Benchmark [14], which is intended to replicate faceted search of bibliographic data. For Barton, the readily available RDF data was derived from

a dump MIT’s Barton library catalog. The original relational data is not available. We replace it with a relational form of DBLP. These benchmarks provide semantically equivalent queries in SQL and SPARQL.

The objective of this evaluation is to observe if commercial relational databases already subsume the optimizations (detection of unsatisfiable conditions and self-join elimination) needed to support effective SPARQL execution on relationally stored data. Therefore the evaluation comprises of a comparison between the execution time of the Ultrawrap SQL queries and the execution time of the native SQL queries on the respective RDBMS. Other possible experiments include comparing Ultrawrap with other RDB2RDF systems, however this is not in the scope of this work. Nevertheless, as shown in the results, Ultrawrap query execution time is comparable with the execution time of native SQL. Such results have not been accomplished by any other RDB2RDF system before [20, 31].

5.1 Platform

Ultrawrap was installed on Microsoft SQL Server 2008 R2 Developer Edition, IBM DB2 9.2 Express Edition and Oracle 11g Release 2 Enterprise Edition. Experiments were conducted on a Sun Fire X4150 with four core Intel Xeon X7460 2.66 GHz processor and 16 GB of RAM running Microsoft Windows Server 2008 R2 Standard on top of VMWare ESX 4.0. SQL Server and Oracle had access to all cores and memory, while DB2 had only access to one core and 2 GB of RAM.

5.2 Workload

The BSBM dataset is equivalent to approximately 100 million triples and requires approximately 11 GB of storage. For Barton, the DBLP dataset is equivalent to approximately 45 million triples and requires approximately 4 GB of storage. Indexes were built on every foreign key and on attributes that were being selected on in the benchmark queries. The execution time was calculated by using the elapsed time returned from SQL Server’s SET STATISTICS ON, DB2’s db2batch and Oracle’s SET TIMING ON option.

Note that the DB2 Express Edition limits itself to 2 GB of RAM. Otherwise, the available RAM is larger than the benchmark databases. To control for this, both cold and warm start experiments were run. Warm start experiments were done by loading the data, restarting the databases and executing variants of each query twenty times. Cold start experiments were done by restarting the database after each execution of a query. The results of the cold start experiments do not add to the paper, because the results are typical of query execution with an empty cache: I/O overhead to retrieve data from disk. The I/O overhead is larger in DB2 because more paging is necessary due to the smaller size of RAM. The average time of warm start experiments is reported. Characteristics of the queries are shown in Table 6.

Table 6. Query characteristics of the BSBM and Barton queries

	Inner Joins	Left-Outer Join	Variable Predicates
Low Selectivity	BSBM 1, 3, 10 Barton 5, 7	-	BSBM 9, 11 Barton 1, 2, 3,
High Selectivity	BSBM 4, 5, 6, 12	BSBM 2, 7, 8	4, 6

From the initial assessment, observations could be organized in four cases:

Case 1) Detection of Unsatisfiable Conditions and Self-join Elimination: if both optimizations are applied then the UNION ALLs of the Tripleviews should not appear in the query plans and redundant self-joins should be eliminated. The execution time and query plans of Ultrawrap queries should be comparable to the corresponding native SQL queries. This should be the case for all queries except variable predicate queries.

Case 2) Detection of Unsatisfiable Conditions without Self-join Elimination: if only the first optimization is applied, then the UNION ALLs of the Tripleviews do not appear in the query plans and the number of subqueries is equal to the number of triple patterns in the original SPARQL query. When the selectivity is high, the execution time of Ultrawrap queries should be comparable to native SQL queries because the amount of tuples that are self-joined is small. On the other hand, when selectivity is low the amount of tuples joined is larger and overhead should be evident. Note that the self-join elimination optimization can only be applied after the UNIONs have been eliminated; hence the complementary case does not occur.

Case 3) No optimizations: If no optimizations are applied then the UNION ALLs of the Tripleviews are not eliminated. In other words, the physical query plan is equal to the initial logical query plan (e.g. Figure 7). The Ultrawrap query execution time should not be comparable to the native SQL queries because every SFW statement in the Tripleviews must be executed.

Case 4) Predicate variable queries: Given the direct mapping, the predicate variable in a SPARQL query is a one-to-many mapping that ranges over all attributes in the database. These types of queries cannot use the mapping between the attributes and its corresponding Tripleview because the attribute is unknown. Further, because the attribute is

unknown, detection of unsatisfiable conditions cannot be applied. For these queries, the Tripleview described in Refinement 1 is used.

In a paper on the use of views in data integration, Krishnamurthy et. al. show that queries with variables ranging over attributes and table names are of higher order logic. Relational algebra languages, such as SQL, do not support higher order logic [33, 34]. Therefore, a SPARQL query with a predicate variable does not have a concise, semantically equivalent SQL query. By concise we mean that the SQL query itself will avoid a union of queries over different tables or columns.

However, with additional domain knowledge, a developer may realize that a SQL query corresponding to a SPARQL query with predicate variables may require only a subset of the tables or attributes of the database. For both BSBM and Barton the original authors exploited their omniscience and wrote concise queries. Thus, it is arguable whether the tests comparing SPARQL queries that contain predicate variables, with the benchmark SQL queries provides a semantically equivalent, apples-to-apples test. Nevertheless, we execute them and include the data.

5.3 Results

Results of two experiments are reported. The first experiment, *Ultrawrap Experiment*, evaluates Ultrawrap implemented as presented. The second experiment, *Augmented Ultrawrap Experiment*, evaluates a version of Ultrawrap augmented with the detection of unsatisfiable conditions optimization.

We determined that DB2 implements both optimizations. SQL Server implements the detection of unsatisfiable conditions optimization. Oracle implements the self-join elimination optimization, but it fails to apply it if the detection of unsatisfiable conditions optimization is not applied. Neither optimization is applied on the predicate variables queries by any RDBMS. Table 7 summarizes the optimizations implemented by each RDBMS. The results of both experiments are presented in Figures 10-12. The Ultrawrap execution time is normalized w.r.t the native SQL execution time for each respective RDBMS, i.e. native SQL execution time is 1.

Table 7. Optimizations implemented by existing RDBMS

RDBMS	Detection of Unsatisfiable Conditions	Self-join Elimination
DB2	Yes	Yes
SQL Server	Yes	No
Oracle	No	Yes

5.3.1 Ultrawrap Experiment

DB2 implements both optimizations. Therefore it is expected that it will execute Ultrawrap queries comparable to native SQL queries (Case 1). This is the case for 7 of the 12 SPARQL queries with bound predicates. For the exceptions, BSBM 1, 3, 4 and Barton 5, the optimizer generated a query plan typical of the native SQL queries, but with a different join order. BSBM 7 has nested left-outer joins. For that query, the DB2 optimizer did not push the respective join predicates into the nested queries and corresponding index-based access paths are not exploited.

SQL Server implements the detection of unsatisfiable conditions optimizations but not self-join elimination. Thus, one would still expect that the high selectivity queries would perform comparable or better than the native SQL queries (Case 2). This is the case for all 7 such queries. For BSBM 4, the optimizer produced a different join order for the two versions of the query, but this time, the Ultrawrap query was better. For the low selectivity queries, review of the query plans reveals the discrepancy in performance is due precisely to the absence of the self-join elimination.

Although Oracle implements self-join elimination it does not apply it in this experiment, and thus does not apply either distinguished optimizations (Case 3). Nevertheless, on 7 of the 12 queries with bound predicates, the Ultrawrap execution is comparable or better than the SQL execution. Review of the query plans yields a third valuable optimization: pushing down selects and join predicates into each of the SFW statements in the UNION ALL of the Tripleviews. Even though each SFW statement is executed, most do not contribute to the final result. By virtue of the additional predicate push-down the execution overhead is minimal.

It is expected that neither optimization be applied for variable predicate queries. This is the case for all three RDBMSs (Case 4). Nevertheless, there are some unanticipated results. The native SQL and Ultrawrap queries for Barton 1 and 6 have similar query plans hence the execution times are comparable. SQL Server outperforms the other systems on BSBM queries 9 and 11 while Oracle outperforms the other systems on Barton 3 and 4. For these cases, the optimizer pushed selects down.

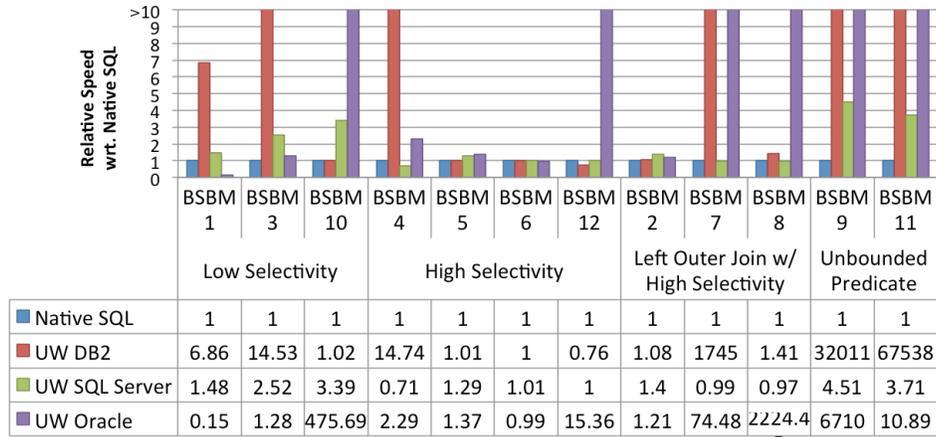


Figure 10. Ultrawrap Experiment results on BSBM.

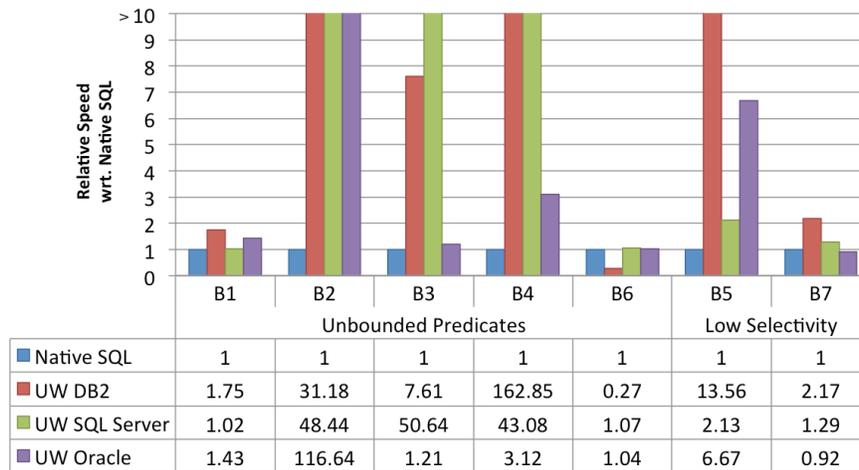


Figure 11. Ultrawrap Experiment results on Barton.

5.3.2 Augmented Ultrawrap Experiment

Augmented Ultrawrap greedily applies the detection of unsatisfiable conditions optimization; that is it applies the optimization on the queries with bound predicates. This should not, and did not impact the behavior of queries with variable predicates. For clarity and space, the corresponding data is omitted. Figure 12 contains the results for the Augmented Ultrawrap experiment.

In this experiment Cases 2 and 3 are eliminated. Of the three RDBMS' only Oracle does not implement detection of unsatisfiable conditions. Thus, despite experimenting with closed proprietary systems, this experiment constitutes a controlled test of the value of this optimization.

Observe that Oracle now performs comparable or better on all bound predicate Ultrawrap queries than the comparable SQL queries. Inspection of the plans reveals that the Oracle optimizer applies the self-join elimination optimization where it did not in the first experiment. Thus, in the second experiment, Oracle's plans include both distinguished transformations (Case 1). SQL Server results are largely unchanged.

The only unanticipated results were changes for DB2 for the unsuccessful bounded predicate queries from the Ultrawrap Experiment (BSBM 1, 3, 4, 7 and Barton 5). In all cases, performance improved. This was the result of changes in the join order, and choosing additional index-based access paths. But in only 1 of the 5 queries, BSBM 1, does the optimizer choose the same join-order as the native SQL query. We investigated a number of options to get better join orders and concomitant theories as to the search behavior of the DB2 optimizer. None of these panned out.

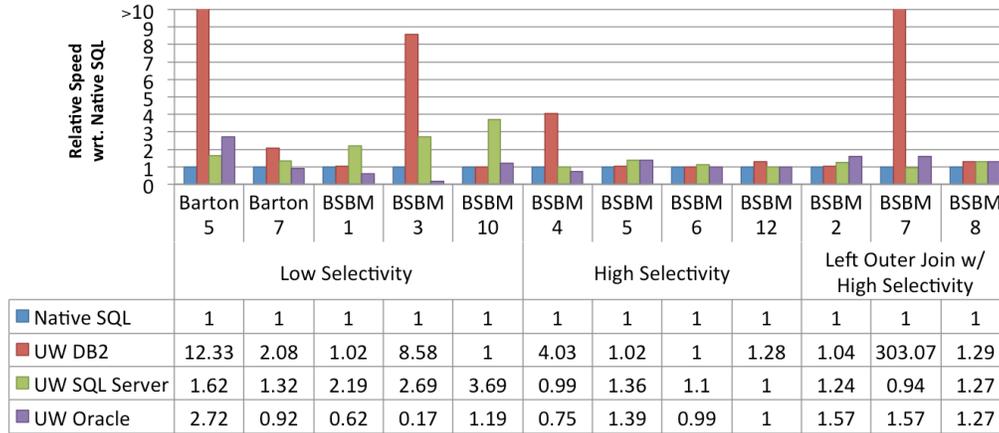


Figure 12. Augmented Ultrawrap Experiment results on BSBM and Barton bounded predicate queries.

6 DISCUSSION

The following points deserve elaboration:

Self-join elimination: The number of self-joins and their elimination is not, by itself, an indicator of poor performance. The impact of the self-join elimination optimization is a function of the selectivity and the number of properties in the SPARQL query that are co-located in a single table. The value of optimization is less as selectivity increases. The number of self-joins in the plan corresponds to the number of properties co-located in a table. The phenomenon is reminiscent of the debate concerning the use of row-stores vs. column stores [12, 21, 44, 47]. Consideration of row-stores vs. column-stores is outside the scope of this paper. Nevertheless, we note that there is debate within the community on the use of row-stores or column-stores for native RDF data and our measurements may help ground that debate [12, 21, 44, 47].

Pushing selects and join predicates: The experiments with Oracle revealed that pushing selects and join predicates could be as effective as the detection of unsatisfiable conditions optimization. For the case of BSBM 7 on Oracle, the optimizer did not push the join predicates down; hence the poor query execution time.

Left-Outer Joins: The experimental results are supportive of hearsay in the Semantic Web community that the endemic use of OPTIONAL in SPARQL queries, which compiles to a left-outer join, is outside the experience of the database community. We speculate that these types of queries are not common in a relational setting, hence the lack of support in commercial systems. Indeed we found that no commercial optimizer eliminates self left-outer joins and OPTIONALS appear in many of the queries where suboptimal join orders are determined.

Join Ordering: Join order is a major factor for poor query execution time, both on Ultrawrap and native SQL queries. Even though DB2 eliminated self-joins in the original Ultrawrap experiment, the optimizer often generated sub-optimal join order for the Ultrawrap queries but did so less often for the Augmented Ultrawrap queries. A possible explanation is simply the size of the search space. For the Ultrawrap queries the optimizer had to evaluate each query within the large union in the definition of the Tripleviews. The Augmented Ultrawrap queries greedily eliminated this potentially explosive component of the search space.

Counting NULLs: Each SFW statement of the Tripleview filters null values. Such a filter could produce an overhead, however we speculate that the optimizer has statistics of null values and avoids the overhead.

To support our hypothesis, it is not necessary for us to produce in a single instance a system that is universally good. However, where and how a system fails to attain an excellent query plan is important, particularly when Semantic Web workloads become common. Given the target RDBMSs are proprietary systems we can only speculate to root causes.

7 CONCLUSION

The context of this research is in an area that has not had much attention: RDB2RDF systems. In addition to moving the focus of RDB2RDF systems to the database, the architecture of Ultrawrap poses other advantages. The relational data is not replicated, thus there is real-time consistency between relational and SPARQL access to the data. By virtue of automatically creating the putative ontology from the SQL-DDL, the system provides a completely automatic method for making legacy commercial databases upward compatible to the Semantic Web. Ultrawrap accesses the database through JDBC. The compile step of Ultrawrap requires database privileges to read the database catalog and

create views, the runtime requires no special privileges. For both halves of Ultrawrap a single implementation is portable across all three vendors.

The experimental results provide evidence supporting our hypothesis, that commercial RDBMSs already subsume the algorithms and optimizations needed to support effective SPARQL execution on existing relationally stored data. It is important to note that our hypothesis does not stipulate that the optimizer will do the right thing every time. In particular, just two optimizing transformations that do appear in commercial RDBMSs are sufficient to render a query plan derived from a SPARQL query to readily appear similar to plans derived from comparable SQL queries. A third optimization, pushing select and join predicates into the argument of a union, enables useful performance improvements across the workload, but does not rewrite the plan into one more typical of a comparable SQL query. Even though Ultrawrap was not compared to other RDB2RDF systems, the results of the experiments show that SPARQL queries on Ultrawrap execute at nearly the same speed as semantically equivalent native SQL queries. These results have not been accomplished by any other RDB2RDF systems [20, 31].

The only point of controversy may be our distinction of SPARQL queries with predicate variables. In these queries, the standard RDB2RDF mapping rules stipulate the variable may be bound to the name of any column in the database. With these semantics, none of the commercial RDBMSs were able to eliminate any elements of union used to define a Tripleview. However, developers familiar with the complete schema of the RDBMS application are able to choose one attribute from each of one or two tables, and write a simple SQL query or a query with a single union.

Queries with predicate variables should not be dismissed as a special case. Queries of this form are intrinsic to faceted search, which is an increasingly common use case. Even so, two arguments that maintain support for our hypothesis include; one, per Krishnamurthy et al [33], predicate variables are a syntactic construct of higher-order logic, therefore the simple SQL queries expressed in the benchmark as equivalent to the SPARQL queries, produce the same answers on the test data, (they are operationally equivalent), but their formal semantics is not the same, and thus should not be used as a basis of comparison. The formally equivalent queries will contain a union [17] and bare comparable performance penalty. A second, more constructive argument is before writing the so-called equivalent SQL query, the SQL developers determined, a priori, which attributes were relevant to the query and which were inconsistent, and they themselves detected unsatisfiable conditions and simply did not code them. Thus, it remains for the community to develop more sophisticated, broadly applicable preconditions for the application of the detection of unsatisfiable conditions transform.

8 ACKNOWLEDGMENTS

We thank Conor Cunningham for his advice on SQL Server and Diego Funes for the implementation of the SPARQL to SQL query translator. This work was supported by the National Science Foundation under grant 1018554. Juan Sequeda was supported by a NSF Graduate Research Fellowship.

REFERENCES

- [1] Berlin SPARQL Benchmark. <http://www4.wiwiw.fu-berlin.de/bizer/BerlinSPARQLBenchmark> .
- [2] D2R Server. <http://www4.wiwiw.fu-berlin.de/bizer/d2r-server/> .
- [3] ODEMapster. <http://neon-toolkit.org/wiki/ODEMapster> .
- [4] Personal communication with Conor Cunningham.
- [5] Quest. <http://obda.inf.unibz.it/protege-plugin/quest/quest.html> .
- [6] RDF Primer. W3C Recommendation. <http://www.w3.org/TR/rdf-primer> .
- [7] RDF Schema. W3C Recommendation <http://www.w3.org/TR/rdf-schema/>.
- [8] Spyder. <http://www.revelytix.com/content/spyder> .
- [9] SquirrelRDF. <http://jena.sourceforge.net/SquirrelRDF> .
- [10] Virtuoso RDF Views. <http://virtuoso.openlinksw.com/whitepapers/relational%20rdp%20views%20mapping.html> .
- [11] W3C OWL Working Group. OWL 2 Web ontology language document overview. W3C Recommendation 27 October 2009, <http://www.w3.org/TR/owl2-overview/>.
- [12] D. Abadi, S. Madden and N. Hachem. Column-stores vs. row-stores: how different are they really? In SIGMOD, pages 967-980, 2008.
- [13] D. Abadi, A. Marcus, S. Madden and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In VLDB, pages 411-422, 2007.
- [14] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Using the Barton libraries dataset as an RDF benchmark. Technical Report MIT-CSAIL-TR-2007-036, MIT.
- [15] R. Angles and C. Gutierrez. The Expressive Power of SPARQL. In ISWC, pages 114-129, 2008.
- [16] M. Arenas, A. Bertails, E. Prud'hommeaux, and J. F. Sequeda. A Direct Mapping of Relational Data to RDF. W3C Candidate Recommendation 23 February 2012, <http://www.w3.org/TR/2012/CR-rdb-direct-mapping-20120223/>.
- [17] F. Barbancon and D.P. Miranker. Implementing Federated Database Systems by Compiling SchemaSQL. In IDEAS, pages 192-201, 2002.
- [18] J. Barrasa, O. Corcho and A. Gomez-Perez. R2O, an Extensible and Semantically based Database-to-Ontology Mapping Language. In SWDB, 2004.
- [19] C. Bizer, T. Heath and T. Berner-Lee. Linked Data - The Story So Far. Int. J. Semantic Web Inf. Syst. 5(3), pages 1- 22, 2009.
- [20] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. Int. J. Semantic Web Inf. Syst. 5(2), pages 1-24, 2009.
- [21] N. Bruno. Teaching an Old Elephant New Tricks. In CIDR, 2009.

- [22] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In ISWC, pages 54–68, 2002.
- [23] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi and D. F. Savo. The MASTRO System for Ontology-based Data Access. *Semantic Web Journal*. 2(1), pages 43-53.
- [24] P. Cangialosi, C. Consoli, A. Faraotti and G. Vetere. Accessing data through ontologies with ONDA. In CASCON, pages 13-26, 2010.
- [25] U. Chakravarthy, J. Grant and J. Minker. Logic-Based Approach to Semantic Query Optimization. *ACM Trans. Database Syst.* 15(2) pages 162-207, 1990.
- [26] A. Chebotko, S. Lu and F. Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data Knowl. Eng.* 68(10), pages 973-1000, 2009.
- [27] Q. Cheng, J. Gryz, F. Koo, T.Y. Cliff Leung, L. Liu, X. Qian and K. Schiefer. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In VLDB, pages 687-698, 1999.
- [28] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An Efficient SQL-based RDF Querying Scheme. In VLDB, pages 1216–1227, 2005.
- [29] S. Das, S. Sundara and R. Cyganiak. R2RML: RDB to RDF Mapping Language. W3C Candidate Recommendation 23 February 2012, <http://www.w3.org/TR/2012/CR-r2rml-20120223/>.
- [30] B. Elliott, E. Cheng, C. Thomas-Ogboji and Z. Ozsoyoglu. A complete translation from SPARQL into efficient SQL. In IDEAS, pages 31-42, 2009.
- [31] A. Gray, N. Gray and I. Ounis. Can RDB2RDF Tools Feasibly Expose Large Science Archives for Data Integration? In ESWC, pages 491-505, 2009.
- [32] B. He, M. Patel, Z. Zhang and K. C-C. Chang. Accessing the deep web. *Commun. ACM*, 50:94–101, May 2007.
- [33] R. Krishnamurthy, W. Litwin and W. Kent. Language Features for Interoperability of Databases with Schematic Discrepancies. In SIGMOD, pages 40-49, 1991.
- [34] L. Lakshmanan, F. Sadri and I. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems. In VLDB, pages 239-250, 1996.
- [35] H. MahmoudiNasab and S. Sakr. An Experimental Evaluation of Relational RDF Storage and Querying Techniques. In DASFAA Workshops, pages 215-226, 2010.
- [36] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19(1), pages 91- 113, 2010.
- [37] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C Recommendation 15 January 2008, <http://www.w3.org/TR/rdf-sparql-query/>.
- [38] S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. Thibodeau, S. Auer, J. Sequeda, A. Ezzat. A Survey of Current Approaches for Mapping of Relational Databases to RDF. W3C RDB2RDF XG Report, 2009.
- [39] M. Schmidt, T. Hornung, N. Kuchlin, G. Lausen and C. Pinkel. An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In ISWC, pages 82-97, 2008.
- [40] J. F. Sequeda, M. Arenas and D. P. Miranker. On Directly Mapping Relational Databases to RDF and OWL. To appear in WWW 2012.
- [41] J. F. Sequeda, R. Depena, S. H. Tirmizi and D. P. Miranker. Ultrawrap: SQL Views for RDB2RDF. In Demo and Posters ISWC 2011.
- [42] J. F. Sequeda, S. Tirmizi, O. Corcho and D. P. Miranker. Survey of Directly Mapping SQL Databases to the Semantic Web. *Knowledge Eng. Review* 26(4), pages 445-486. 2011.
- [43] S. Shenoy and Z. Ozsoyoglu. A System for Semantic Query Optimization. In SIGMOD, pages 181-195, 1987.
- [44] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes and S. Manegold. Column-Store Support for RDF Data Management: not all swans are white. *PVLDB*, pages 1553- 1563, 2008.
- [45] M. Svihla and I. Jelinek. Benchmarking RDF Production Tools. In DEXA, pages 700-709, 2007.
- [46] S. H. Tirmizi, J. F. Sequeda and D. P. Miranker. Translating SQL Applications to the Semantic Web. In DEXA, pages 450- 464, 2008.
- [47] C. Weiss, P. Karras and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB* 1(1), pages 1008-1019, 2008.
- [48] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In SWDB, pages 131–150, 2003.