Ultrawrap: SPARQL Execution on Relational Data
Juan F. Sequeda
Department of Computer Science
University of Texas at Austin
jsequeda@cs.utexas.edu

Daniel P. Miranker
Department of Computer Science
University of Texas at Austin
miranker@cs.utexas.edu

ABSTRACT
The Semantic Web’s promise of web-wide data integration requires the inclusion of legacy relational databases\(^1\), i.e. the execution of SPARQL queries on RDF representation of the legacy relational data. 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 is embodied in a system, Ultrawrap, that encodes a logical representation of the database as an RDF graph using SQL views and a simple syntactic translation of SPARQL queries to SQL queries on those views. Thus, in the course of executing a SPARQL query, the SQL optimizer uses the SQL views that represent a mapping of relational data to RDF, and optimizes its execution. In contrast, related research is predicated on incorporating optimizing transforms as part of the SPARQL to SQL translation, and/or executing some of the query outside the underlying SQL environment.

Ultrawrap is evaluated using two existing benchmark suites that derive their RDF data from relational data through a Relational Database to RDF (RDB2RDF) Direct Mapping and repeated for each of the three major relational database management systems. Empirical analysis reveals two existing relational query optimizations that, if applied to the SQL produced from a simple syntactic translations of SPARQL queries (with bound predicate arguments) to SQL, consistently yield query execution time that is comparable to that of SQL queries written directly for the relational representation of the data. The analysis further reveals the two optimizations are not uniquely required to achieve a successful wrapper system. The evidence suggests effective wrappers will be those that are designed to complement the optimizer of the target database.

1. INTRODUCTION
We postulate that by carefully constructing unmaterialized SQL views\(^2\) to create a logical representation of a legacy relational database as an RDF graph [4], the existing algorithmic machinery in SQL optimizers is already sufficient to effectively execute SPARQL queries [41] on native relational data [39, 49]. Thereby, legacy relational database systems may be made upwardly compatible with the Semantic Web [15], while simultaneously minimizing the complexity of the wrapping system. This is in contrast to related efforts, detailed below, that are predicated on preprocessing and/or optimizing the SQL query before sending it to the SQL optimizer [2, 5, 7].

\(^1\) By legacy, we mean software/data already in wide use such that an organization is not willing to relinquish the investment.

\(^2\) Unmaterialized views are virtual tables that are defined by a query over other tables in the database. They are not stored in the database but can be queried as if they existed. [29].

Figure 1. Taxonomy of RDF Data Management
To clarify the focus of this research, consider the taxonomy in Figure 1. In RDF data management there are efforts that concern Triplestores and those that concern legacy Relational Databases. Triplestores are database management systems whose data model is RDF, and support at least SPARQL execution against the stored contents. Native triplestores are those that are implemented from scratch [18, 40, 53]. RDBMS-backed Triplestores are built by adding an application layer to an existing relational database management system. Within that literature is a discourse concerning the best database schema, SPARQL to SQL query translations, indexing methods and even storage managers, (i.e. column stores vs. row stores) [9, 25, 27, 29, 54]. NoSQL Triplestores are also being investigated as possible RDF storage managers [29, 33, 35]. In all three cases, RDF is the primary data model.

The research herein is concerned with the mapping of legacy relational data with the Semantic Web, a.k.a Relational Database to RDF (RDB2RDF). Within that, the research concerns Wrapper Systems that present a logical RDF representation of relational data that is physically stored in an RDBMS such that no copy of the relational data is made. It follows that some or all of a SPARQL query evaluation is executed by the SQL engine. An alternative is the relational data is extracted from the relational database, translated to RDF, and loaded (ETL) into a triplestore [22].

Since both RDBMS-backed Triplestores and RDB2RDF Wrapper systems involve relational databases and translation from SPARQL to SQL, there is a potential for confusion. The difference is that RDBMS-backed Triplestores translate SPARQL queries to SQL queries that are executed on database schemas that model and store RDF. RDB2RDF Wrapper systems translate SPARQL queries to SQL queries that are executed on legacy database schemas that model and store relational data.

Approximately 70% of websites have relational database back-ends [32]. The sheer number of websites suggests the success of the Semantic Web is tied to maintaining compatibility and consistency with legacy RDBMSs. Wrapper systems enable
Semantic Web applications to coexist with the legacy applications and avoid consistency problems simply by not creating a replicated copy of the data.

In 2008, Angles and Gutierrez showed that SPARQL is equivalent in expressive power to relational algebra [12]. Thus, one might expect the validity of this research’s postulate to be a foregone conclusion. However, in 2009, two studies that evaluated three RDB2RDF wrapper 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 [16, 31].

A motivation for this paper is to resolve the apparent contradiction among the aforementioned papers. Toward that end we have built a system, Ultrawrap[^45]. Ultrawrap is organized as a set of four compilers with the understanding that the SQL optimizer forms one of the four compilers (Section 3 and Figure 2).

In a two-step, off-line process, Ultrawrap defines a SQL view whose query component is a specification of a mapping from the relational data to an RDF triple representation, the Tripleview. In our experiments the Tripleview is not materialized, (i.e. the defining queries are not executed). Thus the view forms a logical specification of the mapping. Note that this view is extremely large, comprising a union of select-from-where queries, at least one query for each column in the relational database. At the onset of the research we first conducted experiments to confirm that such large view definitions would be parsed by RDBMSs without throwing an exception.

At runtime, a third compiler translates an incoming SPARQL query to a SQL query on the Tripleview. The translation is limited to macro-substitution of each logical SPARQL operator with its equivalent SQL operator. This is straightforward as each SPARQL query operator corresponds to an equivalent relational operator [12].

It follows from the SQL standard that an RDBMS must correctly execute the translated SPARQL query. Consequently, the target RDBMS’s SQL system must both use the logical mapping represented in the Tripleview and optimize the resulting query, forming the fourth compiler.

Ultrawrap is evaluated using the three leading RDBMS systems and two benchmark suites, Microsoft SQL Server, IBM DB2 and Oracle RDBMS, and the Berlin and Barton SPARQL benchmarks (Section 5). The SPARQL benchmarks were chosen as a consequence of the fact that they derived their RDF content from a relational source. The Berlin Benchmark provides both SPARQL queries and SQL queries, where each query was derived independently from an English language specification. Since wrappers produce SQL from SPARQL we refer to the benchmark’s SQL queries as benchmark-provided SQL queries. For Barton, the original relational data is not available and the creator of the benchmark did not create separate SPARQL and SQL queries. We located replacement relational data, namely a relational data dump of DBLP and created separate SPARQL and SQL queries derived independently from an English language specification. The benchmark-provided SQL queries have been tuned for use specifically against each benchmark database. We have packaged the new version of Barton for distribution [6].

By using benchmarks containing independently created SPARQL and SQL queries, and considering the effort and maturity embodied in the leading RDBMS’s SQL optimizers, we suppose that the respective benchmark-provided SQL query execution time forms a worthy baseline, and the specific query plans to yield insight into methods for creating wrappers.

Our findings include:

- A mapping of relational data to a Tripleview comprising three columns does not instigate the SQL optimizers to use indexes. The view was refined to reflect physical schema properties. (Section 3)
- Two known query optimizations, detection of unsatisfiable conditions and self-join elimination [21], when applied, not only result in comparable execution times between SPARQL and the benchmark-provided SQL queries with bound predicates, the optimizers will often produce identical query plans. (Section 4)
- In some cases, a third optimizing transform, join predicate push down, can be as effective as the detection of unsatisfiable conditions. (Section 5)
- SPARQL queries containing variables that bind to the predicate position remain troublesome. We relate this problem to an already described problem concerning the use of views in the implementation of data integration systems. (Section 5)
- 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. (Section 6)
- No system, including those that eliminated self equi-joins, eliminated the self left outer joins. The SPARQL optional operator is, by definition, a left outer join. (Section 6)

By starting with a simple wrapper system and evaluating it with sophisticated SQL query optimizers we are able to identify existing, well understood optimization methods that enable wrappers. The results provide a foundation for identifying minimal requirements for effective wrapper systems.

2. RELATED WORK

Per the taxonomy in Figure 1, systems that involve relational databases are RDBMS-backed Triplestores and RDB2RDF systems. We describe three published research efforts concerning RDBMS-backed triplestores [9, 23, 27]. Three RDB2RDF Wrapper systems have been assessed in the literature: D2RQ [2], SquirrelRDF [5] and Virtuoso RDF Views [7].

RDBMS-backed Triplestores store RDF in different database schemas. Many RDBMS-backed Triplestores use the triple table schema: a table with three attributes, containing one row for each triple [25]. Another approach is the property table: a table comprising of one column containing the subject plus one or more columns for predicates that are defined for the subject [54]. Abadi et al. introduced the vertical partitioned table: a table for every unique predicate in the data [9].

Abadi et al argue for the use of column-store based relational systems as the basis of RDBMS-backed triplestores, as compared to more common row-stores. The paper does not address

[^45]: See acknowledgements.
SPARQL to SQL translation. With respect to translation, the paper’s core contribution is the mapping of RDF to a relational schema comprising one table for each predicate value. The resulting tables each contain two columns, the subject and object. The organization is well suited for join processing on a column-store database.

Chebotko et al. present a translation of SPARQL to SQL, where the RDF is modeled as a triple table. They argue that their translation may be composed with additional mappings, enabling their translation to be applied to any relational model of RDF. They reported empirical results for a synthetic RDF test set of 1 million triples. The generated SQL resembles the relational algebra rules used to define the semantics of SPARQL, resulting in “multiple coalesce functions in one projection, null-accepting predicates, and outer union implementations” [23]. The translation is proven to be semantics preserving. Elliot et al. improve upon Chebotko. Chebotko et al.’s methods exploit nested SQL queries. Central to Elliot et al.’s contribution are algorithms that produce flat SQL queries. Their evaluation was also on a triple table schema with datasets between 2-5 million RDF triples [27].

The authors of the three mentioned RDB2RDF wrapper systems have not published a refereed scientific paper describing their rewriting algorithms and optimizations. Open-source code and formulas 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.

Two overlapping, refereed studies do compare the aforementioned RDB2RDF wrapper systems with native SQL execution on the relational database [16, 31]. Bizer & Schultz compared D2R and Virtuoso RDF Views on MySQL [16]. 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 still the most used in the Semantic Web community and no new system has been introduced and evaluated since then. Bizer & Schultz [16], 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 structured data, e.g. pushing the selection conditions to the underlying database system…”

Other RDB2RDF systems go through an ETL process of extracting relational data, translating it to RDF and loading the results into a triplestore [22, 42]. 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 [50]. 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 [43]. In 2010, Mahmoudi Nasab 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 [37]. In conclusion, benchmark-provided SQL queries on relatively stored data outperform any other approach.

Ontology-based data access systems, such MASTRO, ONDA and Quest [19, 20, 38] focus on mapping relational databases to ontologies in order to perform reasoning during query execution. These systems may be of interest to the reader, but do not support SPARQL and fall outside the taxonomy.

3. ULTRAWRAP

The World Wide Web Consortium (W3C) is fostering RDB2RDF systems through standardization efforts [13, 26]. Ultrawrap is compliant with the W3C RDB2RDF Direct Mapping standard which details an RDF graph representation of the relational data. In addition Ultrawrap also translates the relational schema and accompany SQL constraints into an OWL ontology [44, 46, 52].

![Figure 2. Architecture of Ultrawrap](image-url)

Ultrawrap is comprised of four primary components as shown in Figure 2:

1. The translation of a SQL schema, including constraints, to an OWL ontology: the putative ontology (PO) [44, 46, 52].
2. The creation of an intensional 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 the evaluation of the data mappings and concomitant optimization of the query.

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
To define the mapping of the relational data to RDF, the system first identifies an ontological representation of the relational schema. We implement the RDB2RDF direct mapping of Sequeda et al [46, 52], which includes transformation rules for integrity constraints (foreign keys and primary keys). The RDF representation of the relational data is functionally dependent that ontological mapping. The semantics of query execution for this mapping also have foundation [44]. 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 (a.k.a. a join table) 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.

We have found that when a SQL database schema has been created using good data engineering methodology with supporting CASE tools, the synthesized ontology can be quite good. Since the quality of a database’s data modeling is rarely of that high quality, and the meaning of good ontology is subjective, we allay controversy by calling the result collection of ontology declarations a putative ontology (PO). The serialization of the putative ontology, as an OWL file, is not needed to implement the system. However, as both OWL and the creation of the putative ontology are formally defined, for clarity, the exposition of the paper assumes this to be the case.

We chose the Sequeda et al. mapping [44] over the recently ratified W3C RDB2RDF Direct Mapping standard for many reasons. The sufficient reason central to this paper is the semantics of query execution and evaluation for the W3C mapping is not yet done. Note that except for the identification of join tables, the Sequeda et al. mapping subsumes the W3C standard. The additional mappings do not impact the performance evaluation.

Other reasons for choosing the Sequeda et al. mapping follow. The W3C RDB2RDF Direct Mapping standard makes no provision for the publication of meta-data. The specification of an OWL-DL description of the database enables a version of Ultrawrap that augments the database with the RDF representation of the ontology. This equates to linked-data publication of the database’s meta-data, including functional constraints, in a manner consistent with native semantic data sources, i.e. Ultrawrap provides a completely automatic method for making legacy relational databases upward compatible with all layers of the Semantic Web stack. Without such provision, the developers of Semantic Web applications must have a priori knowledge of the contents of a data source, or develop methods capable of normalizing ad-hoc publication of meta-data. The W3C RDB2RDF Direct Mapping standard does not address the inclusion of integrity constraints. Other research shows that the inclusion of these constraints exposes semantics that may critically improve the performance of automatic data integration methods [28, 51].

3.1.2 Creating the Tripleview
The putative ontology is the input to a second compilation step that creates a logical definition of the relational data as RDF and embeds it in a view definition. The pseudo-code for the algorithms appears in Figures 3 – 6. Per the W3C RDB2RDF Direct Mapping standard, concatenating the table name with the primary key value or table name with attribute name creates unique identifiers for subject, predicate and objects. Subsequently, unique identifiers can be appended to a base URI. The SQL Tripleview is comprised of a union of SELECT-FROM-WHERE (SFW) statements. The WHERE clause filters attributes with null values (IS NOT NULL), given that null values are not expressible in RDF.

Due to its simplicity, our starting point is the triple table approach. Even though, studies have shown that storing RDF with the triple table approach in a relational database is easily improved upon [9, 37], this issue is not relevant to Ultrawrap because the relational data is not being materialized in a triple table; instead the relational data is virtually represented as a triple table through unmaterialized views.

Even though our goal is to define a virtual triple table, 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 number 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). Even though all values were cast to varchar, we observed throughout our experiments that the optimizer was still able to apply operators specific for other datatypes (i.e., >, <, etc). 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.

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 3-6. Figure 7 shows the CREATE VIEW statements for 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.

<table>
<thead>
<tr>
<th>Table 1. Example of Product and Producer table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
</tr>
<tr>
<td>Id</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. Logical contents of Tripleview for types</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>Product1</td>
</tr>
<tr>
<td>Product2</td>
</tr>
<tr>
<td>Producer4</td>
</tr>
<tr>
<td>Producer5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Logical contents of Tripleview for varchar(50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>Product1</td>
</tr>
<tr>
<td>Product2</td>
</tr>
<tr>
<td>Producer4</td>
</tr>
<tr>
<td>Producer4</td>
</tr>
<tr>
<td>Producer5</td>
</tr>
<tr>
<td>Producer5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4. Logical contents of Tripleview for int</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>Product1</td>
</tr>
<tr>
<td>Product1</td>
</tr>
<tr>
<td>Product2</td>
</tr>
<tr>
<td>Product2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5. Logical contents of Tripleview for object properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>Product1</td>
</tr>
<tr>
<td>Product2</td>
</tr>
</tbody>
</table>

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 3. Pseudo-code to create a Tripleview for types
3.2.1 **SPARQL to SQL translation**

SPARQL is a graph pattern matching query language [40] that has the form:

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

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_varchar5 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

---

**Figure 4. Pseudo-code to create a Tripleview for Varchar Datatype Properties**

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

**Figure 6. Pseudo-code to create the createTripleview method**

```
CREATE VIEW Triplev
SELECT "Producer"+id as s, id as p, o_id as o_id FROM Producer;
CREATE VIEW Triplev
SELECT "Product"+id as s, id as p, o_id as o_id FROM Product;
CREATE VIEW Triplev
SELECT "Producer"+id as s, id as p, o_id as o_id FROM Producer;
CREATE VIEW Triplev
SELECT "Product"+id as s, id as p, o_id as o_id FROM Product;
CREATE VIEW Triplev
SELECT "Producer"+id as s, id as p, o_id as o_id FROM Producer;
CREATE VIEW Triplev
SELECT "Product"+id as s, id as p, o_id as o_id FROM Product;
CREATE VIEW Triplev
SELECT "Producer"+id as s, id as p, o_id as o_id FROM Producer;
CREATE VIEW Triplev
SELECT "Product"+id as s, id as p, o_id as o_id FROM Product;
```

**Figure 7. CREATE VIEW statements defining the Tripleviews**

```
CREATE VIEW Tripleview_type(s, id, p, o, o_id) AS
SELECT 'Product'+id as s, id as id, "type" as p, "Product" as o, o as o_id FROM Product
UNION ALL
SELECT 'Producer'+id as s, id as id, "type" as p, "Producer" as o, o as o_id FROM Producer;
CREATE VIEW Tripleview_varchar5(s, id, p, o, o_id) AS
SELECT 'Product'+id as s, id as id, "label" as p, label as o, o as o_id FROM Product WHERE label IS NOT NULL
UNION ALL
SELECT 'Producer'+id as s, id as id, "title" as p, title as o, o as o_id FROM Producer WHERE title IS NOT NULL
UNION ALL
SELECT 'Producer'+id as s, id as id, "location" as p, location as o, o as o_id FROM Producer WHERE location IS NOT NULL;
CREATE VIEW Tripleview_int(s, id, p, o, o_id) AS
SELECT 'Product'+id as s, id as id, "pNum1" as p, pNum1 as o, o as o_id FROM Product WHERE pNum1 IS NOT NULL
UNION ALL
SELECT 'Product'+id as s, id as id, "pNum2" as p, pNum2 as o, o as o_id FROM Product WHERE pNum2 IS NOT NULL;
CREATE VIEW Tripleview_object(s, id, p, o, o_id) AS
SELECT 'Product'+id as s, id as id, "Product#producer" as p, "Producer"+prodFk as o, prodFk as o_id FROM Product
```
Tripleview, in this case tripleview_varchar50 \((\text{value})\) can be retrieved.

### 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 [3]. Defining a Tripleview for each datatype expands 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 [21, 24, 47]. 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 [24].

Figure 8 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. We refer the reader to [21, 24]. In this query plan, for each of the triple patterns in the query, the Tripleview is accessed, which a union of all the SFW statements.

#### 4.1.1 Detection of Unsatisfiable Conditions

The idea of this optimization is to determine that a query result is empty by determining, without executing the query. This happens, for example, when a pair of predicate constants are inconsistent [22]. The application of the following transformations eliminates columns from the plan that are not needed to evaluate the SPARQL query.

**Elimination by Contradiction:** Consider a query \(\text{SELECT } * \text{ FROM } \text{R WHERE } A = x \text{ AND } A = y\) such that \(x \neq y\). Then the result of that query is empty. For example, it is clear that the query \(\text{SELECT } * \text{ FROM Product WHERE ProductsID } = 1 \text{ AND ProductsID } = 2\) 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:

- \(\text{UNION ALL } (\text{null}, s, t) = \text{UNION ALL } (s, t)\)
- \(\text{UNION ALL } (\text{null}, t) = t\)

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

```sql
CREATE VIEW Tripleview_varchar50(s,s_id,p,o,o_id) AS
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 "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”:

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

The first SFW statement from Tripleview_varchar50 defines \(p = 'title'\). The query contains \(p = 'label'\). Both predicates cannot be satisfied simultaneously. Given the contradiction, this particular SFW statement of Tripleview_varchar50 can be replaced with 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. Application of elimination by contradiction enables removing, the unnecessary UNION ALL conditions. Thus the combination of the two transformations reduces the Tripleview to precisely the subset of referenced columns. The differences in the query plans in Figures 8 and 9 illustrate the impact of these optimizations.

**Figure 8. Initial query plan of the running example**

**Figure 9. Query plan after application of Detection of Unsatisfiable Conditions optimization**

#### 4.1.2 Augmenting Ultrawrap

The Ultrawrap architecture is readily extended to include the 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. The view definition nested in the SQL query’s FROM clause is replaced with the SFW statement.

### 4.1.3 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 [21]. 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:

```sql
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:

```sql
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:

```sql
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:

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

Figure 10 shows the final query plan after the self-joins are removed.

![Query plan after self-join elimination optimization](image)

**Figure 10. 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, 16] imitates the query load of an e-commerce website. The Barton Benchmark [10] replicates faceted search of bibliographic data. For Barton, the readily available RDF data was derived from a dump of MIT’s Barton library catalog. The original relational data is not available. Similarly the only queries that are available are queries written in SQL against a triple table schema. We have created a version of Barton on par with BSBM and have organized a package so the community may reuse it. In lieu of MIT’s library catalog we used a relational form of DBLP. We deduced the original SPARQL queries and their specification. From the query specification we wrote SQL queries that operate directly on the relational version of DBLP. Details of the relational schemas and queries for the BSBM and Barton benchmark can be found in the Appendix.

The objective of this evaluation is to observe the behavior of commercial relational databases. Therefore the evaluation compares execution time, queries plans, and the optimizing transforms used between the Ultrawrap SQL queries and the benchmark-provided SQL queries on the respective RDBMS. Other possible experiments include comparing Ultrawrap with other RDB2RDF wrappers 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 benchmark-provided SQL queries. Such results have not been accomplished by any other RDB2RDF system [16, 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 a 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 are not qualitatively different than the warm start results and thus are omitted.

The benchmark queries consist of a wide variety of operators and characteristics: Basic Graph Patterns, UNION, FILTER, OPTIONAL, ORDER BY and unbounded predicates with high and low selectivity. Details about the queries for both BSBM and Barton benchmark can be found in the Appendix and on our website. Characteristics of the queries are shown in Table 6.
Table 6. Query characteristics of the BSBM and Barton queries

<table>
<thead>
<tr>
<th></th>
<th>Inner Joins</th>
<th>Left-Outer Join</th>
<th>Predicate Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Selectivity</td>
<td>BSBM 1, 3, 10 Barton 5, 7</td>
<td>-</td>
<td>BSBM 9, 11 Barton 1, 2, 3, 4, 6</td>
</tr>
<tr>
<td>High Selectivity</td>
<td>BSBM 4, 5, 6, 12</td>
<td>BSBM 2, 7, 8</td>
<td></td>
</tr>
</tbody>
</table>

The initial assessment suggests observations be organized as 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 benchmark-provided SQL queries. This should be the case for all queries except the special-case of predicate variable queries, which form Case 4.

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 benchmark-provided SQL queries because the number of tuples that are self-joined is small. On the other hand, when selectivity is low the number of tuples joined is larger and the overhead is more 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 8). The Ultrawrap query execution time should not be comparable to the benchmark-provided SQL queries because every SFW statement in the Tripleviews must be executed.

Case 4) Predicate variable queries: Predicate variable queries are queries that have a variable in the predicate position of a triple pattern. Given an RDB2RDF 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. [34] 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 [34, 36]. Similarly, 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.

For the SPARQL predicate variable queries, when writing the benchmark SQL queries, a SQL developer has visibility on the SQL schema and has related domain knowledge. In most cases that developer will understand that only a few columns are of interest, and write a smaller SQL query than the corresponding SPARQL query. In other words, the SQL query will query certain individual columns, but the SPARQL query will expand to query all columns. This occurs for all such queries for both benchmarks. Thus, it is arguable whether the tests comparing SPARQL queries that contain predicate variables, with the benchmark-provided 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 11-13. The Ultrawrap execution time is normalized w.r.t the benchmark-provided SQL query execution time for each respective RDBMS, i.e. benchmark-provided SQL query execution time is 1.

Table 7. Optimizations implemented by existing RDBMS

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>Detection of Unsatisfiable Conditions</th>
<th>Self-join Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL Server</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Oracle</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 (BSBM 2, 5, 6, 8, 10, 12 and Barton 7). For the exceptions, BSBM 1, 3, 4 and Barton 5, the optimizer generated a query plan typical of the benchmark-provided 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 benchmark-provided 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 benchmark-provided SQL query execution. Review of the query plans yields a third valuable optimization: join predicate push-down 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 predicate variable queries. This is the case for all three RDBMSs (Case 4). Nevertheless, there are some unanticipated results. The benchmark-provided SQL queries 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.

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 benchmark-provided 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 optimizations (Case 1). For BSBM 1, 3, 4 and Barton 7, the Ultrawrap execution is better than the benchmark-provided SQL query execution because the optimizer produced an optimal join order for the Ultrawrap queries. To the best of our knowledge, the benchmark-provided SQL queries were tuned for better performance. Due to lack of Oracle DBA skills, benchmark-provided SQL queries BSBM 1, 3, 4 and Barton 7 were not tuned to the best performance possible.

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. Qualitatively, the result is predictable. The conclusion on quantitative results follows by comparing performance of low selectivity vs. high selectivity queries on SQL Server as shown in Figure 11 and 13. 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 started by Abadi et al [8, 9, 17, 48, 53]. Consideration of row-stores vs. column-stores is outside the scope of this paper. Nevertheless, we note that these measurements may help ground that debate.
Join predicate push-down: The experiments with Oracle revealed that pushing join predicates can 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: 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. 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.

Join Ordering: Join order is a major factor for poor query execution time, both on Ultrawrap and benchmark-provided 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 Ultrawrap queries the optimizer has to evaluate each query within the large union in the definition of the Tripleviews. The Augmented Ultrawrap eliminates unneeded UNION ALL elements, reducing 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.

7. CONCLUSION
RDB2RDF wrapper systems do not replicate relational database content in order to support Semantic Web applications. Architectures that include such wrappers bypass any challenges that form when a database is replicated. Similarly, wrappers provide a low-risk path for existing IT organizations to develop semantic applications. The benefit is transparent in use cases where a semantic application must operate in real-time on data that is being updated by an existing relational database application.

To date, wrapper systems have suffered problems in performance and scalability [16, 31]. Yet, enterprise class relational database systems do not suffer so. Ultrawrap, and the experiments in this paper move the focus to the relational systems. The primary result being that the application of two known semantic query optimizations may yield a query plan typical of a relational query plan, but starting from a logical plan representation of a SPARQL query coupled with a logical plan entailment of a mapping from rows to triples. In this work, we consider SPARQL 1.0 [41].

The research commenced with a hypothesis that not only were such optimizing transforms already known, but they are already implemented in commercial software. To support the hypothesis, it is not necessary to demonstrate that a single system is universally good. Nor does the hypothesis stipulate that the optimizer will do the right thing every time. However, where and how a system failed to attain an excellent query plan is as critical to the analysis as success. Although in some cases, as the target RDBMSs are proprietary systems we can only speculate to root causes. For SPARQL queries with bound predicate arguments the experimental results support the hypothesis. Two key optimizing transformations do appear in commercial RDBMSs, and when applied render a SPARQL query plan comparable to the plan generated for benchmark provided SQL queries. These optimizations are not unique. Experiments reveal a third optimization, join predicate push-down, which pushes join predicates into a view containing unions, enables useful performance improvements across the workload, but does not rewrite the plan into one more typical of a comparable SQL query.

Although we stipulated that tuned SQL query plans for the benchmark provided SQL queries forms a good baseline, we can not rule out the existence of optimizations, perhaps not yet known in the literature, that may provide for further improvement. The third optimization, join predicate push-down underscores the problem is not closed.

Even if one is satisfied with this paper’s existence proposition, the empirical results still demonstrate there is work to be done. Analysis of incongruous performance between benchmark SQL queries and SPARQL queries revealed that relational optimizers do not always determine optimal join orders. This is not news, and even one of the benchmark SQL queries was not optimized correctly. However, this issue manifest more often for the SPARQL queries. We cannot examine the internals of these systems to determine if the complexity of the Ultrawrap queries is challenging the optimizers cost function, the search strategy or both. Recall Ultrawrap transforms a SPARQL query to a SQL query by naively substituting SPARQL operators in the SPARQL query plan with relational operators. Independent of the reason for the optimizers failing to determine optimal join orders, the mere fact that they are failing suggests there is opportunity for improvement by means of less naïve translations.

Even though Ultrawrap was not compared to other RDB2RDF wrapper systems, the results of the experiments show that SPARQL queries with bound predicates on Ultrawrap execute at nearly the same speed as semantically equivalent benchmark-provided SQL queries. These results have not been accomplished by any other RDB2RDF systems [16, 31].

The only point of controversy may be our distinction of SPARQL queries with predicate variables. In these queries, an RDB2RDF mapping stipulates that the variable may be bound to the name of any column in the database. With these semantics, none of the commercial RDBMSs are able to eliminate any elements of the Tripleview union. However, developers familiar with the SQL schema of the RDBMS application are able to choose particular columns from specific tables.

Queries with predicate variables should not be dismissed as a special case. Queries of this form are intrinsic to faceted search, an increasingly common use case. Even so, two arguments that maintain support for our hypothesis include; one, per Krishnamurthy et al [34], 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 [14] and bear comparable performance penalty. A second, more constructive argument is before writing the benchmark-provided 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. In any case, queries with unbound predicate variables remain an open problem.

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. The name of our system, Ultrawrap, was adopted, in part, to pay homage to the late Jack Schwartz who developed the Ultracomputer, an early parallel computer. This work was supported by the National Science Foundation under grant 1018554. Juan Sequeda was supported by a NSF Graduate Research Fellowship.

9. REFERENCES
[3] Personal communication with Conor Cunningham.


