Building an efficient RDF store over a relational database

Mihaela A. Bornea [1], Julian Dolby [2], Anastasios Kementsietsidis [3]
Kavitha Srinivas [4], Patrick Dantressangle [5], Octavian Udrea [6], Bishwaranjan Bhattacharjee [7]
IBM Research
{[1]mbornea, [2]dolby, [3]akement, [4]ksrinivs, [6]oudrea, [7]bhatta}@us.ibm.com, [5]dantress@uk.ibm.com

ABSTRACT

Efficient storage and querying of RDF data is of increasing importance, due to the increased popularity and widespread acceptance of RDF on the web and in the enterprise. In this paper, we describe a novel storage and query mechanism for RDF which works on top of existing relational representations. Reliance on relational representations of RDF means that one can take advantage of 35+ years of research on efficient storage and querying, industrial-strength transaction support, locking, security, etc. However, there are significant challenges in storing RDF in relational, which include data sparsity and schema variability. We describe novel mechanisms to shred RDF into relational, and novel query translation techniques to maximize the advantages of this shredded representation. We show that these mechanisms result in consistently good performance across multiple RDF benchmarks, even when compared with current state-of-the-art stores. This work provides the basis for RDF support in DB2 v.10.1.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

RDF; Efficient Storage; SPARQL; Query optimization

1. INTRODUCTION

While the Resource Description Framework (RDF) [14] format is gaining widespread acceptance (*e.g.,* Best Buy [3], New York Times [18]), efficient management of RDF data is still an open problem. In this paper, we focus on two aspects of efficiency, namely, storage and query evaluation. Proposals for storage of RDF data can be classified into two categories, namely, *Native* stores (*e.g.,* Jena TDB [23], RDF-3X [13], 4store [1]) which use customized binary RDF data representations, and *Relationally-backed* stores (*e.g.,* Jena SDB [23], C-store [2]) which shred RDF data to appropriate relational tables. While there is evidence that going native pays in terms of efficiency, we cannot completely disregard relationally-backed stores. For one thing, relational stores come with 35+ years of research on efficient storage and querying. More importantly, relational-backed stores offer important features that are mostly lacking from native stores, namely, scalability, industrial-strength transaction support, compression, security, to name a few. However, there are important challenges in using a relational database as an RDF store, the most important of which stem from the inherent mismatch between the relational and RDF models. Dynamic RDF schemas and data sparsity are typical characteristics of RDF data which are not commonly associated with relational databases. So, it is not a coincidence that existing approaches that store RDF data over relational stores [2,21,23] cannot handle this dynamicity without altering their schemas. More importantly, existing approaches cannot scale to large RDF stores and cannot handle efficiently many complex queries. Our first contribution is an innovative relational storage representation for RDF data that is both flexible (it does not require schema changes to handle dynamic RDF schemas), and scalable (it handles efficiently the most complex queries).

Efficient querying is our next contribution, with the query language of choice in RDF currently being SPARQL [16]. Although there is a large body of work in query optimization (both in SPARQL [8,11,13,17,19] and beyond), there are still important challenges in terms of (a) SPARQL query optimization, and (b) translation of SPARQL to equivalent SQL queries. Typical approaches perform bottom-up SPARQL query optimization, *i.e.,* individual triples [17] or conjunctive SPARQL patterns [13] are independently optimized, and then the optimizer orders and merges these individual plans into one global plan. These approaches are similar to typical relational optimizers which rely on statistics to assign costs to query plans (in contrast to approaches [19] where statistics are ignored). While these approaches are adequate for simple SPARQL queries, they are not as effective for more complicated, but still common, SPARQL queries, as we illustrate in this paper. Such queries often have deep, nested sub-queries whose inter-relationships are lost when optimizations are limited by the scope of single triple or individual conjunctive patterns. To address such limitations, we introduce a hybrid two-step approach to query optimization. As a first step, we construct a specialized structure, called a data flow, that captures the inherent inter-relationships due to the sharing of common variables or constants of different query components. These inter-relationships often span the boundaries of simple conjuncts (or disjuncts) and are often across the different levels of nesting of a query, *i.e.,* they are not visible to existing bottom-up optimizers. As a second step, we use the data flow and cost estimates to decide both the order with which to optimize the different query components, and the plans we are to consider.

While our hybrid optimizer searches for optimal plans, this search must be qualified by the fact that our SPARQL queries must be converted to SQL. That is, our plans should be such that when they

SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.

are implemented in SQL, they are (a) amenable to optimizations by the relational query engine; and (b) can be efficiently evaluated in the underlying relational store. So in our setting, SPARQL acts as a declarative query language that is optimized, while SQL becomes a procedural implementation language for our plans. This dependence on SQL essentially transforms our problem from a purely query optimization problem into a combined query optimization and translation problem. The translation part is particularly complex since there are many equivalent SQL queries that implement the same SPARQL query plan. Consistently finding the right SQL query is one of the key challenges and contributions of our work.

Note that both the hybrid optimization and the efficient SPARQL-to-SQL translation are contributions that are not specific to our work, and both techniques are generalizable and can be applied in any SPARQL query evaluation system. So our hybrid optimizer can be used for SPARQL query optimization, independent of the selected RDF storage (with or without a relational back-end); our efficient translation of SPARQL to SQL can be generalized and used for any relational storage configuration of RDF (not just the one we introduce here). The combined effects of these two independent contributions drive the performance of our system.

The effectiveness of both our optimizer and our relational backend are illustrated through detailed experiments. There has been lots of discussion [6, 9] as to what is a representative RDF dataset (and associated query workload) for performance evaluation. Different papers have used different datasets with no clear way to correlate results across works. To provide a thorough picture of the current state-of-the-art, and illustrate the novelty of our techniques, we provide as our last contribution an experimental study that contrasts the performance of *five* systems (including ours) against *four* different (real and benchmark) data sets. Our work provides the basis for RDF support in DB2 v.10.1.

2. RDF OVER RELATIONAL

There have been many attempts to shred RDF data into the relational model. One approach involves a single *triple-store* relation with three columns, for the subject, predicate and object. Then, each RDF triple becomes a single tuple, which for a popular dataset like DBpedia results in a relation with 333M tuples (one per RDF triple). Figure 1(a) shows a sample of DBpedia data, used as our running example. The triple-store can deal with dynamic schemas since triples can be inserted without a priori knowledge of RDF data types. However, efficient querying requires specialized techniques [13]. A second alternative is a *type-oriented* approach [23] where one relation is created for each RDF data type. So, for our data in Figure 1(a), we create one relation for people (*e.g.,* to store Charles Flint triples) and another for companies (*e.g.,* to store Google triples). Dynamic schemas require schema changes as new RDF types are encountered, and the number of relations can quickly get out of hand if one considers that DBpedia includes 150K types. Finally, a third alternative [2, 21] considers a *predicate-oriented* approach centered around column-stores where a binary subject-object relation is created for each predicate. So, in our example, we create one relation for the *born*, one for the *died* predicate etc. Similar to the type-oriented approach, dynamic schemas are problematic as new predicates result in new relations, and in a dataset like DBpedia these can number in the thousands. In what follows, we introduce a fourth *entity-oriented* alternative which avoids both the skinny relation of the first approach, and the schema changes (and thousands of relations) required by the latter two.

2.1 The DB2RDF schema

The triple-store offers flexibility in the tuple dimension since new triples irrespectively of type or predicate are added to the relation. The intuition behind our entity-oriented approach is to carry this flexibility in the column dimension. Specifically, the lesson learned from the latter two alternatives is that there is value in storing objects of the same predicate in the same column. So, we introduce a mechanism in which we treat the columns of a relation as flexible storage locations that are not pre-assigned to any predicate, but predicates are assigned to them *dynamically*, during insertion. The assignment ensures that a predicate is always assigned to the same column or more generally the same set of columns.

We describe the basic components of DB2RDF schema in Figure 1. The *Direct Primary Hash (DPH)* (shown in Figure 1(b) and populated with the data from Figure 1(a)) is the main relation in the schema. Briefly, DPH is a wide relation in which each tuple stores a subject s in the *entry* column, with all its associated predicates and objects stored in the $pred_i$ and val_i columns $0 \leq i \leq k$, respectively. If subject s has more than k predicates, *i.e.,* $|pred(s)| > k$, then $((|pred(s)|/k) + 1)$ tuples are used for s, *i.e.,* the first tuple stores the first k predicates for s, and s *spills* (indicated by the *spill* column) into a second tuple and the process continues until all the predicates for s are stored. For example, all triples for Charles Flint in Figure 1(a) are stored in the first DPH tuple, while the second DPH tuple stores all Larry Page triples. Assuming more than k predicates for Android, the third DPH tuple stores the first k predicates while extra predicates, like graphics, spill into the fourth DPH tuple.

Multi-valued predicates require special treatment since their multi-values (objects) cannot fit into a single val_i column. Therefore, we introduce a second relation, called the *Direct Secondary Hash (DS)*. When storing a multi-valued predicate in DPH, a new unique identifier is assigned as the value of the predicate. Then, the identifier is stored in the DS relation and is associated with each of predicate values. To illustrate, in Figures 1(b) and (c), the industry for Google is associated with lid:1 in DPH, while lid:1 is associated in the DS relation with object values Software and Internet.

Note that although a predicate is always assigned to the same column (for any subject having this predicate), the same column stores multiple predicates. So, we assign the founder predicate to column $pred_3$ for both the Charles Flint and the Larry Page subjects, but the same column is also assigned to predicates like kernel and graphics. Having all the instances of a predicate in the same column provides us with all the advantages of traditional relational representations (*i.e.,* each column stores data of the same type) which are also present in the type-oriented and predicate-oriented representations. Storing different predicates in the same column leads to significant space savings since otherwise we would require as many columns as predicates in the data set. In this manner, we use a relatively small number of physical columns to store datasets with a much larger number of predicates. This is also consistent with the fact that although a dataset might have a large number of predicates, not all subjects instantiate all predicates. So, in our sample dataset, the predicate born is only associated with subjects corresponding to humans, like Larry Page, while the founded predicate is associated only with companies. Of course, a key question is how exactly we do this assignment of predicates to columns and how we decide this value k. We answer this question in the Section 2.2 and also provide evidence that this idea actually works in practice.

From an RDF graph perspective, the DPH and DS relations essentially encode the outgoing edges of an entity (the predicate *from* a subject). For efficient access, it is advantageous to also encode the incoming edges of an entity (the predicates *to* an object). To this end, we provide two additional relations, called the *Reverse*

(Charles Flint, born, 1850)
(Charles Flint, died, 1934)
(Charles Flint, founder, IBM)
(Larry Page, born, 1973)
(Larry Page, founder, Google)
(Larry Page, board, Google)
(Larry Page, home, Palo Alto)
(Android, developer, Google)
(Android, version, 4.1)
(Android, kernel, Linux)
(Android, preceded, 4.0)
. . .
(Android, graphics, OpenGL)
(Google, industry, Software)
(Google, industry, Internet)
(Google, employees, 54,604)
(Google, HQ, Mountain View)
(IBM, industry, Software)
(IBM, industry, Hardware)
(IBM, industry, Services)
(IBM, employees, 433,362)
(IBM, HQ, Armonk)

(a) Sample DBpedia data

entry	*spill*	*pred*$_1$	*val*$_1$	*pred*$_2$	*val*$_2$	*pred*$_3$	*val*$_3$	. . .	*pred*$_k$	*val*$_k$
Charles Flint	0	died	1934	born	1850	founder	IBM	. . .	*null*	*null*
Larry Page	0	board	Google	born	1973	founder	Google	. . .	home	Palo Alto
Android	1	developer	Google	version	4.1	kernel	Linux	. . .	preceded	4.0
Android	1	*null*	*null*	*null*	*null*	graphics	OpenGL	. . .	*null*	*null*
Google	0	industry	lid:1	employees	54,604	*null*	*null*	. . .	HQ	Mtn View
IBM	0	industry	lid:2	employees	433,362	*null*	*null*	. . .	HQ	Armonk

(b) Direct Primary Hash (DPH)

l_id	*elm*
lid:1	Software
lid:1	Internet
lid:2	Software
lid:2	Hardware
lid:2	Services

(c) Direct Secondary Hash (DS)

entry	*spill*	*pred*$_1$	*val*$_1$	. . .	*pred*$_{k'}$	*val*$_{k'}$
1850	0	born	Charles Flint	. . .	*null*	*null*
1973	0	born	Larry Page	. . .	*null*	*null*
1934	0	*null*	*null*	. . .	died	Charles Flint
IBM	0	*null*	*null*	. . .	founder	Charles Flint
				. . .		
Software	0	industry	lid:3	. . .	*null*	*null*
Hardware	0	industry	lid:4	. . .	*null*	*null*

(d) Reverse Primary Hash (RPH)

l_id	*elm*
lid:3	IBM
lid:3	Google
lid:4	IBM
lid:4	Google

(e) Reverse Secondary Hash (RS)

Figure 1: Sample DBpedia RDF data and the corresponding DB2RDF schema

Predicate Set	Freq.
SV_1 SV_2 SV_3 SV_4 MV_1 MV_2 MV_3 MV_4	.01
SV_1 SV_2 SV_3 MV_1 MV_2 MV_3	.24
SV_1 SV_3 SV_4 MV_1 MV_3 MV_4	.25
SV_2 SV_3 SV_4 MV_2 MV_3 MV_4	.25
SV_1 SV_2 SV_4 MV_1 MV_2 MV_4	.24
SV_5 SV_6 SV_7 SV_8	.01

Table 1: Micro-Bench Characteristics

Query	Star query predicate set	Results
Q1	SV_1 SV_2 SV_3 SV_4	938
Q2	MV_2 MV_2 MV_3 MV_4	10313
Q3	SV_1 MV_1 MV_2 MV_3 MV_4	10313
Q4	SV_1 SV_2 MV_1 MV_2 MV_3 MV_4	10313
Q5	SV_1 SV_2 SV_3 MV_1 MV_2 MV_3 MV_4	10313
Q6	SV_1 SV_2 SV_3 SV_4 MV_1 MV_2 MV_3 MV_4	10313
Q7	SV_5	2500
Q8	SV_5 SV_6	2500
Q9	SV_5 SV_6 SV_7	2500
Q10	SV_5 SV_6 SV_7 SV_8	2500

Table 2: Micro-Bench Queries

```
SELECT ?s WHERE { ?s SV1 ?o1 . ?s SV2 ?o2 . ?s SV3 ?o3 . ?s SV4 ?o4 }
```

(a) SPARQL for Q1

```
SELECT   T.entry FROM DPH AS T
WHERE    T.PRED0='SV1' AND T.PRED1='SV2' AND T.PRED2='SV3' AND T.PRED3='SV4'
```

(b) Entity-oriented SQL

```
SELECT   T1.SUBJ FROM TRIPLE AS T1, TRIPLE AS T2, TRIPLE AS T3, TRIPLE AS T4
WHERE    T1.PRED='SV1' AND T2.PRED='SV2' AND T3.PRED='SV3' AND T4.PRED='SV4' AND
         T1.SUBJ = T2.SUBJ AND T2.SUBJ = T3.SUBJ AND T3.SUBJ = T4.SUBJ
```

(c) Triple-store SQL

```
SELECT   SV1.ENTRY FROM COL_SV1 AS SV1, COL_SV2 AS SV2, COL_SV3 AS SV3, COL_SV4 AS SV4
WHERE    SV1.ENTRY = SV2.ENTRY AND SV2.ENTRY = SV3.ENTRY AND SV3.ENTRY = SV4.ENTRY
```

(d) Predicate-oriented SQL

Figure 2: SPARQL and SQL queries for Q1

Primary Hash (RPH) and the *Reverse Secondary Hash (RS)*, with samples shown in Figures 1(d) and (e).

Advantages of DB2RDF *Layout.*

An advantage of the DB2RDF schema is the elimination of joins in star queries (i.e., queries that ask for multiple predicates for the same subject or object). Star queries are quite common in SPARQL workloads, and complex SPARQL queries frequently contain subgraphs that are stars. Star queries can involve purely single valued predicates, purely multi-valued predicates, or a mix of both. While for single valued predicates the DB2RDF layout reduces star query processing to a single row lookup in the DPH relation, processing of multi-valued or mixed stars requires additional joins with DS relation. It is unclear how these additional joins impact the performance of DB2RDF when compared to the other types of storage.

To this end, we designed a micro benchmark that contrasts query processing in DB2RDF with the triple-store and predicate-oriented approaches[1]. The benchmark has 1M RDF triples with the characteristics defined in Table 1. Each table row represents a predicate set along with its relative frequency distribution in the data. So, subjects with the predicate set $\{SV_1, SV_2, SV_3, SV_4, MV_1, MV_2, MV_3, MV_4\}$ (first table row) constituted 1% of the 1 million dataset. The predicates SV_1 to SV_8 were all single valued, whereas MV_1 to MV_4 were multi-valued. The predicate sets are such that a single valued star query for SV_1, SV_2, SV_3 and SV_4 is highly selective *but only when all four predicates are involved in the query.* The predicates by themselves are not selective. Similarly, a multi-valued star query for MV_1, MV_2, MV_3 and MV_4 is selective, but only if it involves all four predicates. We also consider a set of selective single valued predicates (SV_5 to SV_8) to separately examine the effects of changing the size of a highly selective single valued star on query processing, while keeping the result set size constant.

Table 2 shows the predicate sets used to construct star queries. Figure 2(a) shows the SPARQL star query corresponding to the predicate set for Q1 in Table 2. For each constructed SPARQL query, we generated three SQL queries, one for each of the DB2RDF, triple-store, and predicate-oriented approaches (see Figure 2 for the SQL queries corresponding to Q1). In all three cases, we only index subjects, since the queries only join subjects. Q1 examines single valued star query processing. As shown in Figure 3, DB2RDF was 12X faster than the triple-store, and 3X faster than the predicate-oriented store (78, 940, and 237 ms respectively). Q2 data shows that this result extends to multi-valued predicates, because of the selectivity gain. DB2RDF outperformed the triple-store by 9X and the predicate-oriented store by 4X (124, 1109 and 426 ms respectively). Q3-Q6 show that the result extends to mixed stars of single and multi-valued predicates, with query times significantly worsening with increased number of conjuncts in the query for the triple-store (1287-1850 ms), while times in the predicate-oriented store show noticeable increases (514-614 ms). In contrast, DB2RDF query times are stable (131-139 ms). Q7-Q10 show a similar trend in the single valued star query case, when any one of the predicates in the star is selective (66-73 ms for DB2RDF, 203-249 for triple-store, and 2-6 ms in the predicate-oriented store). When each predicate involved in the star was highly selective, the predicate-oriented store outperformed DB2RDF. However, DB2RDF is more

[1] We omitted the type-oriented approach because for this micro-benchmark it is similar to the entity-oriented approach.

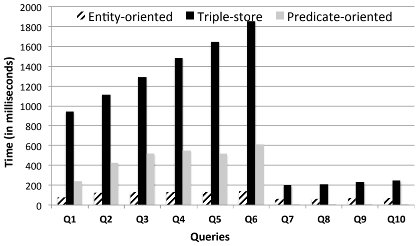


Figure 3: Schema micro-bench results

stable across different conditions (all 10 queries), whereas the performance of the predicate-oriented store depends on predicate selectivities and fluctuates significantly. Overall, these results suggest that DB2RDF has significant benefits for processing generic star queries. Beyond star queries, in Section 4 we show that for a wide set of datasets and queries, DB2RDF is significantly better when compared to existing alternatives.

2.2 Predicate-to-Column assignment

The key to entity-oriented storage is to fit (ideally) all the predicates for a given entity on a single row, while handling the inherent variability of different entities. Because the maximum number of columns in a relational table is fixed, the goal is to dynamically assign each predicate of a given dataset to a column such that:

1. the total columns used across all subjects is minimized.

2. for a subject, mapping two different predicates into the same column (assignment conflict) is minimized to reduce *spills*, since spills cause self-joins, which in turn degrades performance[2].

At an abstract level, a predicate mapping is simply a function that takes an arbitrary predicate p and returns a column number.

Definition 2.1 (Predicate Mapping)**.** A *Predicate Mapping* is a function URI $\rightarrow \mathfrak{N}$ the domain of which is URIs of predicates and the range of which is natural numbers between 0 and an implementation-specified maximum m. Since these mappings are assigning predicates to columns in a relational store, m is typically chosen to be the largest containable on a single database row.

A single predicate mapping function is not guaranteed to minimize spills in predicate insertion, i.e., the mapping of two different predicates of the same entity into the same column. Hence, we introduce predicate mapping compositions to minimize conflicts.

Definition 2.2 (Predicate Mapping Composition)**.** A *Predicate Mapping Composition*, written $f_{m,1} \oplus f_{m,2} \oplus \ldots \oplus f_{m,n}$, defines a new predicate mapping that combines the column numbers from multiple predicate mapping functions $f_1, \ldots f_n$:

$$f_{m,1} \oplus f_{m,2} \oplus \ldots \oplus f_{m,n}(p) \equiv \{v_1, \ldots, v_n \,|f_{m,i}(p) = v_i\}$$

A single predicate mapping function assigns a predicate in exactly one column; so data retrieval is more efficient. However, there are greater possibilities for conflicts, which would force self-joins to gather data across spill rows for the same entity. When predicate composition is used, then the implementation must select a column number in the sequence for predicate insertion and must potentially check all those columns when attempting to read data. This can negatively affect data retrieval, but could reduce conflicts in the data, and eliminate self-joins across multiple spill rows.

We describe two varieties of predicate mapping functions, depending upon whether, or not, a sample of the dataset is available (e.g., due to an initial bulk load, or in the process of data reorganization). If no such sample is available, we use a hash function based on the string value of any URI; when such a sample is available, we exploit the structure of the data sample using graph coloring.

Hashing.

A straightforward implementation of Definition 2.1 is a hash function h_m computed on the string value of a URI and restricted to a range from 0 to m. To minimize spills, we compose n independent hashing functions to provide the column numbers

$$h^n_m \equiv h_{m1} \oplus h_{m2} \oplus \ldots \oplus h_{mn}$$

predicate	h_1	h_2
developer	1	3
version	2	1
kernel	1	3
preceeded	k	1
graphics	3	2

Table 3: Hashes

To illustrate how composed hashing works, consider the Android triples in Figure 1(a) and the two hash functions in Table 3. Further assume these triples are inserted one-by-one, in order, into the database. The first triple (Android, developer, Google) creates a new tuple for subject Android and predicate developer is inserted into $pred_1$, since h_1 puts it there and the column is currently empty. The next triple, (Android, version, 4.1), inserts in the same tuple predicate version in $pred_2$. The third triple, (Android, kernel, Linux), is mapped to $pred_1$ by h_1, but the column is full, so it is inserted into $pred_3$ by h_2. (Android, preceded, 4.0) is inserted into $pred_k$ by h_1. Finally, (Android, graphics, OpenGL) is mapped to column $pred_3$ by h_1 and $pred_2$ by h_2; however, both of these locations are full. Thus, a spill tuple is created which results in the layout shown in Figure 1(b).

Graph Coloring.

When a substantial dataset is available (say, from bulk loading), we exploit the structure of the data to minimize the number of total columns and the number of columns for any given predicate. Specifically, our goal is to ensure that we can overload columns with predicates that do not co-occur together, and assign predicates that do co-occur together to different columns. We do that by creating an interference graph from co-occurring predicates, and use graph coloring to map predicates to columns.

Definition 2.3 (Graph Coloring Problem)**.** A graph coloring problem is defined by an *interference graph* $G =< V, E >$ and a set of *colors* C. Each edge $e \in E$ denotes a pair of nodes in V that must be given different colors. A *coloring* is a mapping that assigns each vertex $v \in V$ to a color different from the color of any adjacent node; note that a coloring may not exist if there are too few colors. More formally,

$$M(G,C) = \left\{ \langle v, c \rangle \;\middle|\; \begin{array}{l} v \in V \wedge \\ c \in C \wedge \\ (\langle v_i, c_i \rangle \in M \wedge \langle v, v_i \rangle \in E \rightarrow c \neq c_i) \end{array} \right\}$$

Minimal coloring would be ideal for predicate mapping, but to be useful the coloring must have no more colors than the maximum number of columns. Since computing a truly minimal coloring is NP-hard in general, we use the Floyd-Warshall greedy algorithm to approximate a minimal coloring.

To apply graph coloring to predicate mapping, we formulate an interference graph consisting of edges linking every pair of predicates that both appear in any subject. That is, we create $G_D =< V_D, E_D >$ for an RDF dataset D where

$$V_D = \{p \,|< \mathtt{s}, \mathtt{p}, \mathtt{o} >\in \mathtt{D}|\}$$
$$E_D = \{< p_i, p_j > |< \mathtt{s}, \mathtt{p_i}, \mathtt{o} > \in \mathtt{D} \wedge < \mathtt{s}, \mathtt{p_j}, \mathtt{o} > \in \mathtt{D}|\}$$

If a coloring $M(G_D, C)$ such that $|C| \leq m$ exists, then it provides a mapping of each predicate to precisely one database column. We use c^D_m to be a predicate mapping defined by coloring

[2] The triple store illustrates this point clearly since it can be thought of as a *degenerate* case of R^{2}DF that uses a single $pred_i$, val_i column pair and where naive evaluation of queries always requires self-joins.

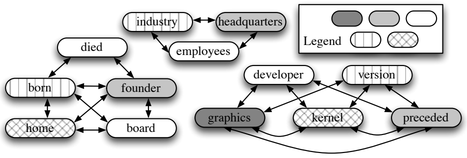


Figure 4: Graph Coloring Example

of dataset D with m or fewer colors. All of our datasets (see Section 4 for details on them) except DBpedia could be colored to fit on a database row. When a coloring does not exist, as in DBpedia, this means there is no way to put all the predicates into the columns such that every entity can be placed on one row and each predicate for the entity be given exactly one column. In this case, we can color a subset of predicates (e.g., based on query workload and the most frequently occurring predicates), and compose a predicate mapping function based on this coloring and hash functions.

We define more formally what we mean by coloring for a subset. Specifically, we define a subset P of the predicates in dataset D, and we write $D \otimes P$ to be all triples in D that have a predicate from P. If we choose P such that the remaining data is colorable with $m-1$ colors, then we define a mapping function

$$\hat{c}_m^{D\otimes P} \equiv \begin{cases} c_{m-1}^{D\otimes P} & p \in P \\ m & p \notin P \end{cases}$$

This coloring function can be composed with another function to handle the predicates not in P, for instance $\hat{c}_m^{D\otimes P} \oplus h_m$. With this predicate mapping composition, we were able to fit most of the data for a given entity on a single row, and reduce spills, while ensuring that the number of columns usage was minimized. Note that this same compositional approach can be used to handle dynamicity in data. If a new predicate p gets added after coloring, the second hash function is used to specify the column assignment for p.

Figure 4 shows how coloring works for the data in Figure 1(a). Predicates died, born, and founder have interference edges because they co-occur for entity Charles Flint. Similarly, founder, born, home and board co-occur for Larry Page and are hence connected. Notice further that the coloring algorithm will color board and died the same color even though both are predicates for the same type of entity (e.g., Person) because they never co-occur together (in the data). Overall, for the 13 predicates, we only need 5 colors.

2.3 Graph Coloring in practice

We evaluated the effectiveness of coloring using four RDF datasets (see Section 4 for details). Our datasets were chosen so that they covered a wide range of skews and distributions [6]. So, for example, while the average out-degree in DBpedia is 14, in LUBM and SP2B it's 6. The average in-degree in DBpedia is 5, in SP2B 2 and in LUBM 8. Beyond averages, out-degrees and in-degrees in DBpedia follow a power-law distribution [6] and therefore some subjects have significantly more predicates than others.

The results of graph coloring for all datasets are shown in Table 4. For the first three datasets, coloring covered 100% of the dataset, and reduced from 30% to as much as 85% the number of columns required in the DPH and RPH relations. So, while the LUBM dataset has 18 predicates, we only require 10 columns in the DPH and 3 in the RPH relations. In the one case where coloring could not cover all the data, it could still handle 94% of the dataset in DPH with 75 columns, and 99% of the dataset in RPH with 51 columns, when we focused on the frequent predicates and the query workload. To put this in perspective, a one-to-one mapping from predicates to columns would require 53,796 columns for DBpedia (instead of 75 and 51, respectively).

We now discuss spills and nulls. Ideally, we want to eliminate spills since they affect query evaluation. Indeed, by coloring in full the first three datasets, we have *no spills* in the DPH and RPH relations. So, storing 100M triples from LUBM in DB2RDF results in 15,619,640 tuples in DPH (one per subject) and 11,612,725 tuples in RPH (one per object). Similarly, storing 100M triples of SP2B in DB2RDF results in 17,823,525 in DPH and 47,504,066 tuples in RPH, without any spills. In DBpedia, storing 333M triples results in DPH and RPH relations with 23,967,748 and 78,697,637 tuples, respectively, with only 808,196 spills in the former (3.37% of the DPH) and 35,924 spills in the latter (0.04% of RPH). Of course, our coloring considered the full dataset before loading so it is interesting to investigate how successful coloring is (in terms of spills) if only a subset of the dataset is considered. Indeed, we tried coloring only 10% of the dataset, using random sampling of records. We used the resulted coloring from the sample to load the full dataset and counted any spills along the way. For LUBM, by only coloring 10% of the records, we were still able to load the whole dataset without any spills. For SP2B, loading the full dataset resulted in a negligible number of spills, namely, 139 spills (out of 17,823,525 entries) in DPH, and 666 (out of 47,504,066 entries) in RPH. More importantly, for DPpedia we only had 222,423 additional spills in DPH (a 0.9% increase in DPH) and 216,648 additional spills in RPH (a 0.3% increase). So clearly, our coloring algorithm performs equally well for bulk and for incremental settings.

Dataset	Triples	Total Predicates	DPH Columns	Percent. Covered	RPH Columns	Percent. Covered
SP2Bench	100M	78	54	100%	53	100%
PRBench	60M	51	35	100%	9	100%
LUBM	100M	18	10	100%	3	100%
DBpedia	333M	53,976	75	94%	51	99%

Table 4: Graph Coloring Results

In any dataset, each subject does not instantiate all predicates, and therefore even in the compressed (due to coloring) DPH and RPH relations not all subjects populate all columns. Indeed, our statistics show that for LUBM, in the DPH relation 64.67% of its predicate columns contain NULLs, while this number is 94.77% for the RPH relation. For DBpedia, the corresponding numbers are 93% and 97.6%. It is interesting to see how a high percentage of NULLs affects storage and querying. In terms of storage, existing commercial (*e.g.,* IBM DB2) and open-source (*e.g.,* Postgres) database systems can accommodate large numbers of NULLs with small costs in storage, by using *value compression*. Indeed, this claim is also verified by the following experiment. We created a 1M triples dataset in which each triple in the dataset had the same 5 predicates and loaded this dataset in our DB2RDF schema using IBM DB2 as our relational back-end. The resulting DPH relation has 5 predicate columns and no NULL values (as expected) and its size on disk was approximately 10.1MB. We altered the DPH relation and introduced (i) 5 additional null-populated predicate/value columns, (ii) 45 null-populated columns, or (iii) 95 null-populated columns. The storage requirements for these relations changed to 10.4MB, 10.65MB and 11.4MB respectively. So, increasing by 20-fold the size of the original relation with NULLs only required 10% of extra space.

We also evaluated these queries across all these relations. The impact of NULLs is more noticeable here. We considered both fast queries with small result sets, and longer running queries with large result sets. The 20-fold increase in NULLs resulted in differences in evaluation times that ranged from as low as 10% to as much as a two-fold increase on the fastest queries. So, while the presence of NULLs has small impact in storage, it can noticeably affect query performance, at least for very fast queries. This illustrates the value of our coloring techniques. By reducing both the number of columns with nulls, and the number of nulls in existing columns, we improve query evaluation and minimize space requirements.

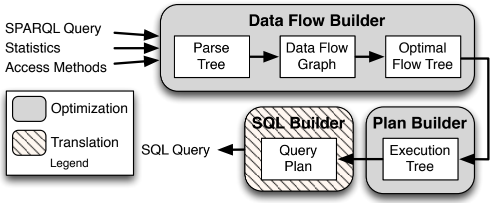


Figure 5: Query optimization and translation architeture

3. QUERYING RDF

Relational systems have a long history of query optimization, so one might suppose that a naive translation from SPARQL to SQL would be sufficient, since the relational optimizer can optimize the SQL query once the translation has occurred. However, as we show empirically here and in Section 4, huge performance gains can occur when SPARQL and the SPARQL to SQL translation are independently optimized. In what follows, we first present a novel hybrid SPARQL query optimization technique, which is generic and independent of our choice of representing RDF data (in relational schema, or otherwise). In fact these techniques can be applied directly to query optimization for native RDF stores. Then, we introduce query translation techniques tuned to our schema representation. Figure 5 shows the steps of the optimization and translation process, as well as the key structures constructed at each step.

3.1 The SPARQL Optimizer

There are three inputs to our optimization:

1. The query $\mathcal{Q}$: The SPARQL query conforms to the SPARQL 1.0 standard. Therefore, each query $\mathcal{Q}$ is composed of a set of hierarchically nested graph patterns $\mathcal{P}$, with each graph pattern $P \in \mathcal{P}$ being, in its most simple form, a set of triple patterns.

2. The statistics $\mathcal{S}$ **over the underlying** RDF **dataset**: The types and precision with which statistics are defined is left to specific implementations. Examples of collected statistics include the total number of triples, average number of triples per subject, average number of triples per object, and the top-k URIs or literals in terms of number of triples they appear in, etc.

3. The access methods $\mathcal{M}$: Access methods provide alternative ways to evaluate a triple pattern t for some pattern $P \in \mathcal{P}$. The methods are system-specific, and dependent on existing indexes. For example, for a system like DB2RDF with only subject and object indexes (no predicate indexes), the methods would be *access-by-subject* (acs), by *access-by-object* (aco) or a *full scan* (sc).

Figure 6 shows a sample input where query $\mathcal{Q}$ retrieves the people that founded or are board members of companies in the software industry. For each such company, the query retrieves the products that were developed by it, its revenue, and optionally its number of employees. The statistics $\mathcal{S}$ contain the top-k constants like *IBM* or *industry* with counts of their frequency in the base triples. Three different access methods are assumed in $\mathcal{M}$, one that performs a data scan (sc), one that retrieves all the triples given a subject (acs), and one that retrieves all the triples given an object (aco).

The optimizer consists of two modules, namely, the *Data Flow Builder* DFB, and the *Query Plan Builder* QPB.

- **Data Flow Builder** (DFB): Query triple patterns typically share variables, and hence the evaluation of one is often dependent on that of another. For example, in Figure 6(a) triple pattern t_1 shares variable $?x$ with both triples patterns t_2 and t_3. In DFB, we use sideways information passing to construct an optimal flow tree, that considers cheaper patterns (in terms of estimated variable bindings) first before feeding these bindings to more expensive patterns.
- **Query Plan Builder** (QPB)**:** While the DFB considers information passing irrespectively of the query structure (*i.e.,*, the nesting of patterns and pattern operators), the QPB module incorporates this structure to build an execution tree (a storage-independent query plan). Our query translation (Section 3.2) uses this execution tree to produce a storage specific query plan.

```
SELECT  ★
WHERE { ?x home “Palo Alto” (t1)
        { ?x founder ?y (t2) UNION
          ?x member ?y (t3) }
        { ?y industry “Software” (t4)
          ?z developer ?y (t5)
          ?y revenue ?n (t6) }
        OPTIONAL {
            ?y employees ?m (t7) } }
```

(a) Sample query $\mathcal{Q}$

value	count
IBM	7
industry	6
Google	5
Software	2
Avg triples per subject	5
Avg triples per object	1
Total triples	26

(b) top-k stats in $\mathcal{S}$

$$\mathcal{M} = \{ sc, acs, aco \}$$

(c) Access methods $\mathcal{M}$

Figure 6: Sample input for query optimization/translation

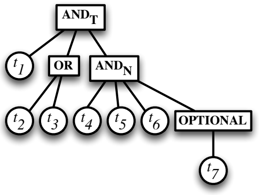


Figure 7: Query parse tree

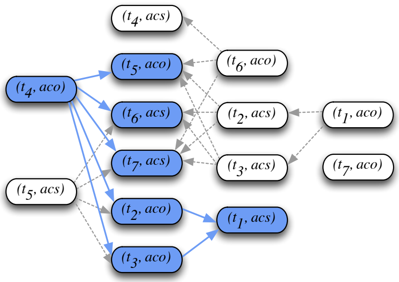


Figure 8: Data flow graph

3.1.1 *The Data Flow Builder*

The DFB starts by building a parse tree for the input query. The tree for the query in Figure 6(a) is shown in Figure 7. Then it uses sideways information passing to compute the data flow graph which represents the dependences amongst the executions of each triple pattern. A node in this graph is a pair of a triple pattern and an access method; an edge denotes one triple producing a shared variable that another triple requires. Using this graph, DFB computes the optimal flow tree (the blue nodes in Figure 7) which determines an optimal way (in terms of minimizing costs) to traverse all the triple patterns in the query. In what follows, we describe in detail how all these computations are performed.

Computing cost.

Definition 3.1 (Triple Method Cost)**.** Given a triple t, an access method m and statistics $\mathcal{S}$, function $\mathsf{TMC}(t, m, \mathcal{S}) :\rightarrow c, c \in \mathbb{R}_{\leq 0}$ assigns a cost c to evaluating t using m wrt statistics $\mathcal{S}$.

The cost estimation clearly depends on the statistics $\mathcal{S}$. In our example, $\mathsf{TMC}(t_4, aco, \mathcal{S}) = 2$ because the exact lookup cost using the object Software is known. For a scan method, $\mathsf{TMC}(t_4, sc, \mathcal{S}) = 26$, *i.e.,* the total number of triples in the dataset. Finally, $\mathsf{TMC}(t_4, acs, \mathcal{S}) = 5$, *i.e.,* the average number of triples per subject, assuming subject is bound by a prior triple access.

Building the Data Flow Graph.

The data flow graph models how using the current set of bindings for variables can be used to access other triples. In modeling this flow, we need to respect the semantics of AND, OR and OPTIONAL patterns. We first introduce a set of helper functions that are used to define the graph. We use ↑ to refer to parents in the query tree structure: for a triple or a pattern, it is the immediately enclosing pattern. We use $*$ to denote transitive closure.

Definition 3.2 (Produced Variables)**.** $\mathcal{P}(t, m) :\rightarrow \mathcal{V}_{prod}$ maps a triple and an access method pair to a set of variables that are bound after the lookup, where t is a triple, m is an access method, and $\mathcal{V}_{prod}$ is the set of variables.

In our example, for the pair $(t4, aco)$, $\mathcal{P}(t_4, aco) :\rightarrow y$, because the lookup uses *Software* as an object, and the only variable that gets bound as a result of the lookup is y.

Definition 3.3 (Required Variables)**.** $\mathcal{R}(t, m) :\rightarrow \mathcal{V}_{req}$ maps a triple and an access method pair to a set of variables that are required to be bound for the lookup, where t is a triple, m is an access method, and $\mathcal{V}_{req}$ is the set of variables.

Back to the example, $\mathcal{R}(t_5, aco) :\rightarrow y$. That is, if one uses the *aco* access method to evaluate t_5, then variable y must be bound by some prior triple lookup.

Definition 3.4 (Least Common Ancestor)**.** $\mathsf{LCA}(p, p')$ is the first common ancestor of patterns p and p'. More formally, it is defined as follows:

$$\begin{array}{l}\mathsf{LCA}(p, p') = x \iff \\ \quad x \in \uparrow^*(p) \wedge x \in \uparrow^*(p') \wedge \nexists y.y \in \uparrow^*(p) \wedge y \in \uparrow^*(p') \wedge x \in \uparrow^*(y)\end{array}$$

As an example, in the Figure 7, the least common ancestor of AND_N and OR is AND_T.

Definition 3.5 (Ancestors To LCA)**.** $\upuparrows(p, p')$ refers to the set of $\uparrow^*$ built from traversing from p to the $\mathsf{LCA}(p, p')$:

$$\upuparrows(p, p') \equiv \{x \,|x \in \uparrow^*(p) \wedge \neg x \in \uparrow^*(\mathsf{LCA}(p, p'))\}$$

For instance, for the query shown in Figure 7, $\upuparrows(t_1, \mathsf{LCA}(t_1, t_2)) = \{\mathsf{AND}_T, \mathsf{OR}\}$

Definition 3.6 (OR Connected Patterns)**.** $\cup$ denotes that two triples are related in an OR pattern, i.e. their least common ancestor is an OR pattern: $\cup(t, t') \equiv \mathsf{LCA}(t, t')$ is OR.

In the example, t_2 and t_3 are $\cup$.

Definition 3.7 (OPTIONAL Connected Pattern)**.** OPTIONAL Connected Patterns $\hat{\cap}$ denotes if one triple is optional with respect to another, i.e. there is an OPTIONAL pattern guarding t' with respect to t:

$$\hat{\cap}(t, t') \equiv \; \exists p : p \in \upuparrows(t', t) \wedge p \text{ is } \mathsf{OPTIONAL}$$

In the example, t_6 and t_7 are $\hat{\cap}$, because t_7 is guarded by an OPTIONAL in relation to t_6.

Definition 3.8 (Data Flow Graph)**.** The *Data Flow Graph* is a graph of $G =< V, E >$, where $V = (\mathcal{T} \times \mathcal{M}) \cup \mathsf{root}$, where root is a special node we add to the graph. A directed edge $(t, m) \rightarrow (t', m')$ exists in V when the following conditions hold:

$$\mathcal{P}(t, m) \supset \mathcal{R}(t', m') \wedge \neg \left(\cup(t, t') \vee \hat{\cap}(t', t)\right)$$

In addition, a directed edge from root exists to a node (t, m) if $\mathcal{R}(t, m) = \emptyset$.

In the example , a directed edge root $\rightarrow (t_4, aco)$ exists in the data flow graph (in Figure 8 we show the whole graph but for simplicity in the figure we omit the root node), because t_4 can be accessed by an object with a constant, and it has no required variables. Further, $(t_4, aco) \rightarrow (t_2, aco)$ is part of the data flow graph, because (t_2, aco) has a required variable y that is produced by (t_4, aco). In turn, (t_2, aco) has an edge to (t_1, acs), because (t_1, acs) has a required variable x which is produced by (t_2, aco).

The *Data Flow Graph* $\mathcal{G}$ is weighted, and the weights for each edge between two nodes is determined by a function:

$$\mathsf{W}((t, m), (t', m')), \mathcal{S}) :\rightarrow w$$

The w is derived from the costs of the two nodes, i.e., $\mathsf{TMC}(t, m, \mathcal{S})$, and $\mathsf{TMC}(t', m', \mathcal{S})$. A simple implementation of this function, for example could apply the cost of the target node to the edge. In the example, for instance, w for the edge root $\rightarrow (t_4, aco)$ is 2, whereas the edge root $\rightarrow (t_4, asc)$ is 5.

Computing The Optimal Flow Tree.

Given a weighted data flow graph $\mathcal{G}$, we now study the problem of computing the optimal (in terms of minimizing the cost) order for accessing *all* the triples in query $\mathcal{Q}$.

Theorem 3.1. *Given a data flow graph $\mathcal{G}$ for a query $\mathcal{Q}$, finding the minimal weighted tree that covers all the triples in $\mathcal{Q}$ is NP-hard.*

The proof is by reduction from the TSP problem and is omitted here due to lack of space. In spite of this negative result, one might think that the input query $\mathcal{Q}$ is unlikely to contain a large number of triples, so an exhaustive search is indeed possible. However, even in our limited benchmarks, we found this solution to be impractical (e.g., one of our queries in the tool integration benchmark had 500 triples, spread across 100 OR patterns). We therefore introduce a greedy algorithm to solve the problem: Let T denote the *execution tree* we are trying to compute. Let τ refer to the set of triples corresponding to nodes already in the tree:

$$\tau \equiv \{t_i \,|\exists m_i \;\; (t_i, m_i) \in \mathcal{T}\}$$

We want to add a node that adds a new triple to the tree while adding the cheapest possible edge; formally, we want to choose a node (t', m') such that

$$\begin{array}{l}(t', m') \in V \wedge \; \textit{\# node to add} \\ t' \notin \tau \wedge \; \textit{\# node adds new triple} \\ \exists(t, m) : \\ \left(\begin{array}{l}(t, m) \in \mathcal{T} \wedge \; \textit{\# node currently in tree} \\ (t, m) \rightarrow (t', m') \wedge \; \textit{\# valid edge to new node} \\ \textit{\# no similar pair of nodes such that...} \\ \nexists(t'', m''), (t''', m''') : \\ \left(\begin{array}{l}(t'', m'') \in \mathcal{T} \wedge \\ t''' \notin \tau \wedge \\ (t'', m'') \rightarrow (t''', m''') \wedge \\ \textit{\# ...adding } (t''', m''') \textit{ is cheaper} \\ \mathsf{W}((t'', m''), (t''', m''')) < \mathsf{W}((t, m), (t', m'))\end{array}\right)\end{array}\right)\end{array}$$

On the first iteration,$\mathcal{T}_0 = \mathsf{root}$, and $\tau_0 = \emptyset$. $\mathcal{T}_{i+1}$ is computed by applying the step defined above, and the triple of the chosen node is added to τ_{i+1}. In our example, root $\rightarrow (t_4, aco)$ is the cheapest edge, so $\mathcal{T}_1 = (t_4, aco)$, and $\tau_0 = t_4$. We then add (t_2, aco) to $\mathcal{T}_2$, and so on. We stop when we get to $\mathcal{T}_n$, where n is the number of triples in $\mathcal{Q}$. Figure 8 shows the computed tree (marked blue nodes) while Figure 9 shows the algorithm, where function $triple(j)$ returns the triple associated with a node in $\mathcal{G}$.

3.1.2 The Query Plan Builder

Both the data flow graph and the optimal flow tree largely ignore the query structure (the organization of triples into patterns) and the operators between the (triple) patterns. Yet, they provide useful information as to how to construct an actual plan for the input query, the focus of this section and output of the QPB module.

In more detail, Figure 10 shows the main algorithm ExecTree of the module. The algorithm is recursive and takes as input the optimal flow tree F computed by DFB, and (the parse tree of) a pattern P, which initially is the main pattern that includes the whole query

```
Input: The weighted data flow graph $\mathcal{G}$
Output: An optimal flow tree $T$
$\tau \leftarrow \emptyset$;
$\mathcal{T} \leftarrow$ root;
$E \leftarrow$ SortEdgesByCost($\mathcal{G}$);
while $|\mathcal{T}| < |\mathcal{Q}|$ do
  for each edge $e_{ij} \in E$ do
    if $i \in \mathcal{T} \wedge j \notin \mathcal{T} \wedge triple(j) \notin \tau$ then
      $\mathcal{T} \leftarrow \mathcal{T} \cup j$;
      $\tau \leftarrow \tau \cup$ triple($j$);
      $T \leftarrow e_{ij}$;
```

Figure 9: The algorithm for computing the optimal flow tree

```
Input: The optimal flow tree $F$ of query $\mathcal{Q}$, a pattern $P$ in $\mathcal{Q}$
Output: An execution tree $T$ for $P$, a set $\mathcal{L}$ of execution sub-trees
1  $T \leftarrow \emptyset$; $\mathcal{L} \leftarrow \emptyset$;
2  switch the type of pattern $P$ do
3    case $P$ is a SIMPLE pattern
4      for each triple pattern $t_i \in P$ do
5        $T_i \leftarrow$ GetTree($t_i$, $F$); $\mathcal{L}_i \leftarrow \emptyset$;
6        if isLeaf($T_i$, $F$) then $\mathcal{L} \leftarrow \mathcal{L} \cup T_i$ ;
7        else ($T$, $\mathcal{L}$) $\leftarrow$ AndTree($F$, $T$, $\mathcal{L}$, $T_i$, $\mathcal{L}_i$)
8    case $P$ is an AND pattern
9      for each sub-pattern $P_i \in P$ do
10       ($T_i$, $\mathcal{L}_i$) $\leftarrow$ ExecTree($F$, $P_i$);
11       ($T$, $\mathcal{L}$) $\leftarrow$ AndTree($F$, $T$, $\mathcal{L}$, $T_i$, $\mathcal{L}_i$);
12   case $P$ is an OR pattern
13     for each sub-pattern $P_i \in P$ do
14       ($T_i$, $\mathcal{L}_i$) $\leftarrow$ ExecTree($F$, $P_i$);
15       ($T$, $\mathcal{L}$) $\leftarrow$ OrTree($F$, $T$, $\mathcal{L}$ $T_i$, $\mathcal{L}_i$);
16   case $P$ is an OPTIONAL pattern
17     ($T'$, $\mathcal{L}'$) $\leftarrow$ ExecTree($F$, $P$);
18     ($T$, $\mathcal{L}$) $\leftarrow$ OptTree($F$, $T$, $\mathcal{L}$ $T'$, $\mathcal{L}'$);
19   case $P$ is a nested pattern
20     ($T$, $\mathcal{L}$) $\leftarrow$ ExecTree($F$, $P$);
21 return ($T$, $\mathcal{L}$)
```

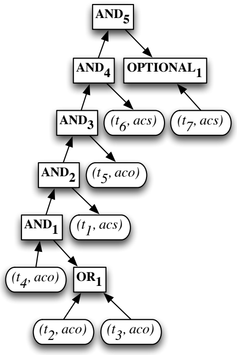


Figure 10: The ExecTree Algorithm and resulting execution tree

$\mathcal{Q}$. So in our running example, for the query in Figure 6(a), the algorithm takes as input the parse tree in Figure 7 and the optimal flow tree in Figure 8. The algorithm returns a schema-independent plan T, called the *execution tree* for the input query pattern P. The set of returned execution sub-trees $\mathcal{L}$ is guaranteed to be empty when the recursion terminates, but contains important information that the algorithm passes from one level of recursion to the previous one(s), while the algorithm runs (more on this later).

There are four main types of patterns in SPARQL, namely, *SIMPLE*, AND, UNION (a.k.a OR), and OPTIONAL patterns, and the algorithm handles each one independently, as we illustrate through our running example. Initially, both the execution tree T and the set $\mathcal{L}$ are empty (line 1). Since the top-level node in Figure 7 is an AND node, the algorithm considers each sub-pattern of the top-level node and calls itself recursively (lines 8-10) with each of the sub-patterns as argument. The first sub-pattern recursively considered is a *SIMPLE* one consisting of the single triple pattern t_1. By consulting the flow tree F, the algorithm determines the optimal execution tree for t_1 which consists of just the node (t_1, acs) (line 5). By further consulting the flow (line 6) the algorithm determines that node (t_1, acs) is a leaf node in the optimal flow and therefore it's evaluation depends on the evaluation of other flow nodes. Therefore, the algorithm adds tree (t_1, acs) to the local *late fusing* set $\mathcal{L}$ of execution trees. Set $\mathcal{L}$ contains execution sub-trees that should not be merged yet with the execution tree T but should be considered later in the process. Intuitively, late fusing plays two main roles: (a) it uses the flow as a guide to identify the proper point in time to fuse the execution tree T with execution sub-trees that are already computed by the recursion; and (b) it aims to optimize query evaluation by minimizing the size of intermediate results computed by the execution tree, and therefore it only fuses sub-trees at the latest possible place, when either the corresponding sub-tree variables are needed by the later stages of the evaluation, or when the operators and structure of the query enforce the fuse. The first recursion terminates by returning $(T_1, \mathcal{L}_1) = (\emptyset, \{L_1 = (t_1, acs)\})$. The second sub-pattern in Figure 7 is an OR and is therefore handled in lines 12-15. The resulting execution sub-tree contains three nodes, an OR node as root (from line 15) and nodes (t_2, aco) and (t_3, aco) as leaves (recursion in line 14). This sub-tree is also added to local set $\mathcal{L}$ and the second recursion terminates by returning $(T_2, \mathcal{L}_2) = (\emptyset, \{L_2 = \{\mathsf{OR}, (t_2, aco), (t_3, aco)\}\})$. Finally, the last sub-pattern in Figure 7 is an AND pattern again, which causes further recursive calls in lines 8-11. In the recursive call that processes triple t_4 (lines 5-7), the execution tree node (t_4, aco) is the root node in the flow and therefore it is merged to the main execution tree T. Since T is empty, it becomes the root of the tree T. The three sub-trees that include nodes (t_5, aco), (t_6, acs), and $OPT = \{(\mathsf{OPTIONAL}), (t_7, aco)\}$ are all becoming part of set $\mathcal{L}$. Therefore, the third recursion terminates by returning $(T_3, \mathcal{L}_3) = ((t_4, aco), \{L_3 = \{(t_5, aco)\}, L_4 = \{(t_6, acs)\}, L_5 = \{(\mathsf{OPTIONAL}), (t_7, aco)\}\}$. Notice that after each recursion ends (line 10), the algorithm considers (line 11) the returned execution T_i and late-fuse $\mathcal{L}_i$ trees and uses function AndTree to build a new local execution T and set $\mathcal{L}$ of late-fusing trees (by also consulting the flow and following the late-fusing guidelines on postponing tree fusion unless it is necessary for the algorithm to progress). So, after the end of the first recursion and the first call to function AndTree, $(T, \mathcal{L}) = (T_1, \mathcal{L}_1)$, *i.e.,*, the trees returned from the first recursion. After the end of the second recursion, and the second call to AndTree, $(T, \mathcal{L}) = (\emptyset, \mathcal{L}_1 \cup \mathcal{L}_2)$. Finally, after the end of the third recursion, $(T, \mathcal{L}) = ((t_4, aco), \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3)$. The last call to AndTree builds the tree to the right of Figure 10 in the following manner. Starting from node (t_4, aco), it consults the flow and picks from the set $\mathcal{L}$ the sub-tree L_2 and connects this to node (t_4, aco) by adding a new AND node as the root of the tree. Sub-trees L_3 , L_4 and L_5 can be added at this stage to T but they are not considered as they violate the principles of late-fusing (their respective variables are not used by any other triple, as is also obvious by the optimal flow). On the other hand, there is still a dependency between the latest tree T and L_1 since the selectivity of t_1 can be used to reduce the intermediate size of the query results (especially the bindings to variable $?y$). Therefore, a new AND is introduced and the existing T is extended with L_1. The process iterates in this fashion until the whole tree in Figure 10 is generated.

Note that by using the optimal flow tree as a guide, we are able to *weave* the evaluation of different patterns, while our structured-based processing guarantees that the associativity of operations in the query is respected. So, our optimizer can generate plans like the one in Figure 10 where only a portion of a pattern is initially evaluated (*e.g.,* node (t_4, aco)) while the evaluation of other constructs in the pattern (*e.g.,* node (t_5, aco)) can be postponed until it no longer can be avoided. At the same time, this de-coupling from query structure allow us to safely push the evaluation of patterns early in the plan (*e.g.,* node (t_1, acs)) when doing so improves selectivity and reduces the size of intermediate results.

3.2 The SPARQL to SQL Translator

The translator takes as input the execution tree generated from the QPB module and performs two operations: first, it transforms the execution tree into an equivalent query plan that exploits the entity-oriented storage of DB2RDF; second, it uses the query plan to create the SQL query which is executed by the database.

3.2.1 Building the Query Plan

The execution tree provides an access method and an execution order for each triple but assumes that each triple node is evaluated

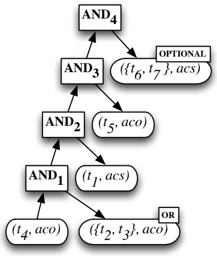
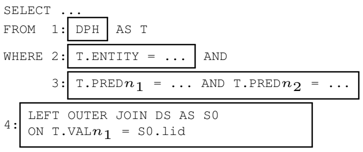


Figure 11: The query plan tree



```
SELECT ...
FROM  1: DPH  AS T
WHERE 2: T.ENTITY = ...  AND
      3: T.PREDn1 = ... AND T.PREDn2 = ...
4: LEFT OUTER JOIN DS AS S0
   ON T.VALn1 = S0.lid
```

Figure 12: SQL code template

independently of the other nodes. However, one of the advantages of the entity-oriented storage is that a single access to, say, the DPH relation might retrieve a row that can be used to evaluate multiple triple patterns (star-queries). To this end, starting from the execution tree the translator builds a query plan where triples with the same subject (or the same object) are *merged* in the same plan node. A merged plan node indicates to the SQL builder that the containing triples form a star-query and must be executed with a single SQL select. Merging of nodes is always advantageous with one exception: when the star query involves entities with spills. The presence of such entities would require self-joins of the DPH (RPH) relations in the resulting SQL statement. Self-joins are expensive and therefore we use the following strategy to avoid them: When we *know* that star-queries involve entities with spills, we choose to *cascade* the evaluation of the star-query by issuing multiple SQL statements, each evaluating a subset of the star-query while at the same time filtering entities from the subsets of the star-query that have been previously evaluated. The multiple SQL statements are such that no SQL statement accesses predicates stored into different spill rows. Of course, the question remains on how we determine whether spills affect a star query. In our system this is straightforward. With only a tiny fraction of predicates involved in spills (due to coloring – see Section 4), our optimizer consults an in-memory structure of predicates involved in spills to determine during merging whether any of the star-query predicates participate in spills.

During the merging process we need to respect both the *structural* and *semantic* constraints. The structural constraints are imposed by the entity-oriented representation of data. To satisfy the structural constraints, candidate nodes for merging need to refer to the same entity, have the same access method and do not involve spills. As an example, in Figure 8 nodes t_2 and t_3 refer to the same entity (due to variable $?x$) and the same access method aco, as do nodes t_6 and t_7, due to the variable $?y$ and the method acs.

Semantic constraints for merging are imposed by the control structure of the SPARQL query (*i.e.,* the AND, UNION, OPTIONAL patterns). This restricts the merging of triples to constructs for which we can provide the equivalent SQL statements to access the relational tables. Triples in conjunctive and disjunctive patterns can be safely merged because the equivalent SQL semantics are well understood. Therefore, with a single access we can check whether the row includes the non-optional predicates in the conjunction. Similarly, it is possible to check the existence of any of the predicates mentioned in the disjunction. More formally, to satisfy the semantic constraints of SPARQL, candidate nodes for merging need to be ANDMergeable, ORMergeable or OPTMergeable.

Definition 3.9 (AND Mergeable Nodes)**.** Two nodes are ANDMergeable iff their least common ancestor and all intermediate ancestors are AND nodes:

$$\text{ANDMergeable}(t,t') \iff \\ \forall x : x \in (\uparrow\uparrow(t, \text{LCA}(t,t')) \cup \uparrow\uparrow(t', \text{LCA}(t,t'))) \implies x \text{ is AND}$$

Definition 3.10 (OR Mergeable Nodes)**.** Two nodes are ORMergeable iff their least common ancestor and all intermediate ancestors are OR nodes:

$$\text{ORMergeable}(t,t') \iff \\ \forall x : x \in (\uparrow\uparrow(t, \text{LCA}(t,t')) \cup \uparrow\uparrow(t', \text{LCA}(t,t'))) \implies x \text{ is OR}$$

Going back to the execution tree in Figure 10, notice that ORMergeable (t_2, t_3) is true, but ORMergeable (t_2, t_5) is false.

Definition 3.11 (OPTIONAL Mergeable Nodes)**.** Two nodes are OPTMergeable iff their least common ancestor and all intermediate ancestors are AND nodes, except the parent of the higher order triple in the execution plan which is OPTIONAL:

$$\text{OPTMergeable}(t,t') \iff \\ \forall x : x \in (\uparrow\uparrow(t, \text{LCA}(t,t')) \cup \uparrow\uparrow(t', \text{LCA}(t,t'))) \\ \implies x \text{ is AND } \vee \{x \text{ is OPTIONAL } \wedge x \text{ is parent of } t'\}$$

As an example, in Figure 10 OPTMergeable (t_6, t_7) is true.

Given the input execution tree, we identify pairs of nodes that satisfy *both* the structural and semantic constraints introduced, and we merge them. Due to lack of space, we omit here the full details of the node merging algorithm and illustrate the output of the algorithm for our running example. So, given as input the execution tree in Figure 10, the resulting query plan tree is shown in Figure 11. Notice that in the resulting query plan there are two node merges, one due to the application of the ORMergeable definition, and one by the application of the OPTMergeable definition. Note that each merged node is annotated with the corresponding semantics under which the merge was applied. As a counter-example, consider node (t_5, aco) which is compatible structurally with the new node $(\{t_2, t_3\}, aco)$ since they both refer to the same entity through variable $?y$, and have the same access method aco. However, these two nodes are not merged since they violate the semantic constraints (*i.e.,* they do not satisfy the definitions above since their merge would mix a conjunctive with a disjunctive pattern). Even for our simple running example, the two identified node merges result in significant savings in terms of query evaluation. Intuitively, one can think of these two merges as eliminating two extra join operations during the translation of the query plan to an actual SQL query over the DB2RDF schema, the focus of our next section.

3.2.2 The SQL *Generation*

SQL generation is the final step of query translation. The query plan tree plays an important role in this process, and each node in the query plan tree, be it a triple, merge or control node, contains the necessary information to guide the SQL generation. For the generation, the SQL builder performs a post order traversal of the query plan tree and produces the equivalent SQL query for each node. The whole process is assisted by the use of SQL code templates.

In more detail, the base case of SQL translation considers a node that corresponds to a single triple or a merge. Figure 12 shows the template used to generate SQL code for such a node. The code in box 1 sets the target of the query to the DPH or RPH tables, according to the access method in the triple node. The code in box 2 restricts the entities being queried. As an example, when the subject is a constant and the access method is acs, the *entry* is connected to the constant subject values. When the subject is variable and the method is acs, then *entry* is connected with a previously-bound variable from a prior SELECT sub-query. The same reasoning applies for the *entry* component for an object when the access method is aco. Box 3 illustrates how one or more predicates are selected. That is, when the plan node corresponds to a merge, multiple $pred_i$ components are connected through conjunctive or disjunctive SQL operators. Finally, box 4 shows how we do outer join with the secondary table for multi-valued predicates.

```
WITH QT_4RPH AS
 SELECT T.val_1 AS val_1 FROM RPH AS T WHERE T.entry ='Software' AND T.pred_1 ='industry',
QT_4DS AS
 SELECT COALESCE (S.elm, T.val_1 ) AS y
 FROM    QT_4RPH AS T LEFT OUTER JOIN DS AS S ON T.val_1 =S.l_id
QT_23RPH AS
 SELECT QT_4DS.y,
         CASE T.pred_m ='founder' THEN val_m ELSE null END AS val_m,
         CASE T.pred_0 ='member' THEN val_0 ELSE null END AS val_0
 FROM    RPH AS T, QT_4DS
 WHERE  T.entry =QT_4DS.y AND (T.pred_m ='founder' OR T.pred_0 ='member'),
QT_23 AS
 SELECT LT.val_0 AS x, T.y FROM QT_23RPH as T, TABLE (T.val_m , T.val_0) as LT(val_0)
 WHERE  LT.val_0 IS NOT NULL
QT_1DPH AS
 SELECT T.entry AS x, QT_23.y FROM DPH AS T, QT_23
 WHERE  T.entry =QT_23.x AND T.pred_k ='home' AND T.val_1 ='Palo Alto',
QT_5RPH AS
 SELECT T.entry AS y, QT_1DPH.x FROM RPH AS T, QT_1DPH
 WHERE  T.entry =QT_1DPH.y AND T.pred_1 ='developer',
QT_67DPH AS
 SELECT T.entry AS y, QT_5RPH.x, CASE T.pred_k ='employees' THEN val_k ELSE null END as z
 FROM    DPH AS T, QT_5RPH WHERE T.entry =QT_5RPH.y AND T.pred_m ='revenue'
SELECT x, y, z FROM QT_67DPH
```

Figure 13: Generated SQL for SPARQL Query in Figure 6

The operator nodes in the query plan are used to guide the connection of instantiated templates like the one in Figure 12. We have already seen how AND nodes are implemented through the variable binding across triples as in box 2. For OR nodes we use the SQL UNION operator to connect its components' previously defined SELECT statements. For OPTIONAL we use LEFT OUTER JOIN between the SQL template for the main pattern and the SQL template for the OPTIONAL pattern. Figure 13 shows the final SQL for our running example where the SQL templates described above are instantiated according to the query plan tree in Figure 11.

In Figure 13, several Common Table Expressions (CTEs) are used for each plan node. t_4 is evaluated first and accesses RPH using the Software constant. Since industry is a multivalued predicate, the RS table is also accessed. The remaining predicates in this example are single valued and the access to the secondary table is avoided. The ORMergeable node t_{23} is evaluated next using the RPH table where the object is bound to the values of y produced by the first triple. The WHERE clause enforces the semantic that at least one of the predicates is present. The CTE projects the values corresponding to the present predicates and null values for those that are missing. The next CTE just flips these values, creating a new result record for each present predicate. The plan continues with triple t_5 and is completed with node the OPTMergeable node t_{67}. Here no constraint is imposed for the optional predicate making its presence optional on the record. In case the predicate is present, the corresponding value is projected, otherwise null. In this example, each predicate is assigned to a single column. When predicates are assigned to multiple columns, the position of the value is determined with CASE statements as seen in the SQL sample.

3.3 Advantages of the SPARQL Optimizer

To examine the effectiveness of our query optimization, we conducted experiments using both our 1M triple microbenchmark of Section 2.1 (which offers more control) and queries from the datasets used in our main experimental section (Section 4). As an example, for our microbenchmark we considered two constant values O_1 and O_2 with relative frequency in the data of .75 and .01, respectively. Then, we issued the simple query shown in Figure 14(a) that allowed data flows in either direction; i.e., evaluation could start on t_1 with an aco using O_1, then use the bindings for ?s to access t_2 with an acs, or start instead on t_2 with an aco using O_2 and use bindings for ?s to access t_1. The latter case is of course better. Figure 14(b) shows the SQL generated by our SPARQL optimizer while Figure 14(c) shows an equivalent SQL query corresponding to the only alternative but sub-optimal flow. The former query took 13 ms to evaluate, whereas the latter took 5X longer, that is 65 ms, suggesting that our optimization is in fact effective even in this simple query. Using real and benchmark queries from datasets resulted in even more striking differences in evaluation times. For example, when optimized by our SPARQL optimizer query, PQ1 from PRBench (Section 4) was evaluated in 4ms, while the translated SQL corresponding to a sub-optimal flow required 22.66 seconds!

```
SELECT ?s WHERE { ?s SV_1 O_1 (t_1) . ?s SV_2 O_2 (t_2) }
```

(a) SPARQL Query

```
SELECT T.ENTRY, D.ENTRY FROM RS AS R, DPH AS D
WHERE   R.ENTRY='O_2' AND R.PROP='SV_2' AND D.ENTRY=T.ENTRY AND D.VAL0='O_1' AND D.PROP0='SV_1'
```

(b) Optimized SQL

```
SELECT T.ENTRY, D.ENTRY FROM RS AS R, DPH AS D
WHERE R.ENTRY='O_1' AND R.PROP='SV_1' AND D.ENTRY=T.ENTRY AND D.VAL0='O_2' AND D.PROP0='SV_2'
```

(c) Alternative SQL

Figure 14: Query Translation

4. EXPERIMENTS

We compared the performance of DB2RDF, using IBM DB2 as our relational back-end, to that of Virtuoso 6.1.5 OpenSource Edition, Apache Jena 2.7.3 (TDB), OpenRDF Sesame 2.6.8, and RDF-3X 0.3.5. DB2RDF, Virtuoso and RDF-3X were run in a client server mode on the same machine and all other systems were run in process mode. For both Jena and Virtuoso, we enabled all recommended optimizations. Jena had the BGP optimizer enabled. For Virtuoso we built all recommended indexes. For DB2RDF, we only added indexes on the entry columns of the DPH and RPH relations (no indexes on the $pred_i$ and val_i columns).

We conducted experiments with 4 different benchmarks: LUBM [7], SP2Bench [15], DBpedia [12], and a private benchmark PRBench that was offered to us by an external partner organization. For the LUBM and SP2Bench benchmarks, we scaled them up to 100 million triples each and used their associated published query workloads. The DBpedia 3.7 benchmark [5] has 333 million triples. The private benchmark included data from a tool integration application, and it contained 60 million triples about various software artifacts generated by different tools (e.g., bug reports, requirements, etc). For all systems, we evaluated queries in a warm cache scenario. For each dataset, benchmark queries were randomly mixed to create a run, and each run was issued 8 times to the 5 stores. We discarded the first run and reported the average result for each query over 7 consecutive runs. For each query, we measured its running time excluding the time taken to stream back the results to the API, in order to minimize variations caused by the various APIs available. As shown in Figure 15, the evaluated queries were classified into four categories. Queries that failed to parse SPARQL correctly, we reported as *unsupported*. The remainder supported queries were further classified as either *complete*, *timeout*, or *error*. We counted the results from each system and when a system provided the correct number of answers we classified the query as completed. If the system returned the wrong number of results, we classified this as an error. Finally, we used a timeout of 10 minutes to trap queries that do not terminate within a reasonable amount of time. In the figure, we also report the average time taken (in seconds) to evaluate complete and timeout queries. For queries that timeout, their running time was set to 10 minutes. For obvious reasons, we do not count the time of queries that return the wrong number of results.

This is the most comprehensive evaluation of RDF systems. Unlike previous works, this is the first study that evaluates 5 systems using a total of *78 queries*, over a total of *600 million triples*. Our experiments were conducted on 5 identical virtual machines (one per system), each equivalent to a 4-core, 2.6GHz Intel Xeon system with 32GB of memory running 64-bit Linux. Each system was not memory limited, meaning it could consume all of its 32G. None of the systems came close to this memory limit in any experiment.

4.1 The datasets

- **LUBM:** The LUBM benchmark requires OWL DL inference, which is not supported across all tested systems. Without inference, most benchmark queries return empty result sets. To address this issue, we expanded the existing queries and created a set of *equivalent* queries that implement inference and do not require this feature from the evaluated system. As an example, if the LUBM ontology stated that $GraduateStudent \sqsubseteq Student$, and the query asks for *?x rdf:type Student*, the query was expanded into *?x rdf:type Student UNION ?x rdf:type Graduate Student*. We performed this set of expansions and issued the same expanded query to all systems. From the 14 original queries in the benchmark, only 12 (denoted as LQ1 to LQ10, LQ13 and LQ14) are included here because 2 queries involved ontological axioms that cannot be expanded.
- **SP2Bench:** SP2Bench is an extract of DBLP data with corresponding SPARQL queries (denoted as SQ1 to SQ17). We used this benchmark as is, with no modifications. Prior reports on this benchmark were conducted with at most 5 million triples (even in the paper where the benchmark was introduced). We scaled to 100 million triples, and noticed that some queries (by design) had rather large result sets. SQ4 in particular created a cross product of the entire dataset, which meant that *all* systems timeout on this query.
- **DBpedia:** The DBpedia SPARQL benchmark is a set of query templates derived from actual query logs against the public DBpedia SPARQL endpoint [12]. We used these templates with the DBpedia 3.7 dataset, and obtained 20 queries (denoted as DQ1 to DQ20) that had non-empty result sets. Since templates were derived for an earlier DBpedia version, not all result in non-empty queries.
- **PRBench:** The private benchmark reflects data from a tool integration scenario where specific information about the same software artifacts are generated by different tools, and RDF data provides an integrated view on these artifacts across tools. This is a quad dataset where triples are organized into over 1 million 'graphs'. As we explain, this caused problems for some systems which do not support quads (e.g., RDF-3X, Sesame). We had 29 SPARQL queries (denoted as PQ1 to PQ29), with some being fairly complex queries (e.g., a SPARQL union of 100 conjunctive queries).

4.2 Experimental results

Main Result 1. Figure 15 shows that DB2RDF is the only system that evaluates correctly and efficiently 77 out of the 78 tested queries. As mentioned, SQ4 was the only query in which our system did timeout (as did all the other systems). If we exclude SQ4, it is clear from Figure 15 that each of the remaining systems had queries returning incorrect number of results, or queries that timeout without returning any results. We do not emphasize the advantage of DB2RDF in terms of SPARQL support, since this is mostly a function of system maturity and continued development.

Main Result 2. Given Figure 15, it is hard to make direct system comparisons. Still, when the DB2RDF system is compared with systems that can evaluate approximately the same queries (*i.e.,* Virtuoso and Jena), then DB2RDF is in the worst case slightly faster, and in the best case, as much as an order of magnitude faster than the other two systems. So, for LUBM, DB2RDF is significantly faster than Virtuoso (2X) and Jena (4X). For SP2Bench, DB2RDF is on average times about 50% faster than Virtuoso, although Virtuoso has a better geometric mean (not shown due to space constraints), which reflects Virtuoso being much better on short running queries. For DBpedia, DB2RDF and Virtuoso have comparable performance, and for PRBench, DB2RDF is about 5.5X better than Jena. Jena is actually the only system that supports the same queries as DB2RDF, and across all datasets DB2RDF is in the worst case 60%, and in the best case as much as two orders of magnitude faster. A comparison between DB2RDF and RDF-3X is also possible, but only in the LUBM dataset where both systems support a similar number of queries. The two systems are fairly close in performance and out-perform the remaining three systems. When compared between themselves across 11 queries (RDF-3X did not run one query), DB2RDF is faster than RDF-3X in 3 queries, namely in LQ8, LQ13 and LQ14 (246ms, 14ms and 4.6secs versus 573ms, 36ms and 9.5secs, respectively), while RDF-3X has clearly an advantage in 3 other queries, namely in LQ2, LQ6, LQ10 (722ms, 12secs and 1.57secs versus 20secs, 33secs and 3.42secs, respectively). For the remaining 5 queries, the two systems have almost identical performance with RDF-3X being faster than DB2RDF by approximately 3ms for each query.

Dataset	System	Supported			Unsupported	Mean
		Complete	Timeout	Error		(secs)
LUBM (100M triples) (12 queries)	Jena	12	-	-	-	35.1
	Sesame	4	-	8	-	164.7
	Virtuoso	12	-	-	-	16.8
	RDF-3X	11	-	-	1	2.8
	DB2RDF	12	-	-	-	8.3
SP2Bench (100M triples) (17 queries)	Jena	11	6	-	-	253
	Sesame	8	8	1	-	330
	Virtuoso	16	1	-	-	211
	RDF-3X	6	2	2	7	152
	DB2RDF	16	1	-	-	108
DBpedia (333M triples) (20 queries)	Jena	18	1	1	-	33
	Virtuoso	20	-	-	-	0.25
	DB2RDF	20	-	-	-	0.25
PRBench (60M triples) (29 queries)	Jena	29	-	-	-	5.7
	Virtuoso	25	-	-	4	3.9
	DB2RDF	29	-	-	-	1.0

Figure 15: Summary results for all systems and datasets

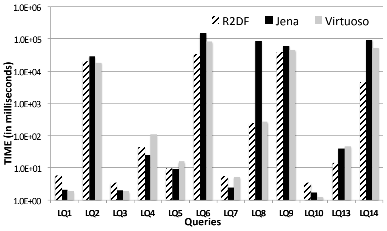


Figure 16: LUBM benchmark results

Detailed results: For a more detailed per-query comparison, we turn to Figure 16 which illustrates the running times for DB2RDF, Virtuoso and Jena for all 12 LUBM queries (reported times are in milliseconds and the scale is logarithmic). Notice that DB2RDF outperforms the other systems in the long-running and complicated queries (*e.g.,* LQ6, LQ8, LQ9, LQ13, LQ14). So, DB2RDF takes approximately 34secs to evaluate LQ6, while Virtuoso requires 83.2secs and Jena 150secs. Similarly, DB2RDF takes 40secs to evaluate LQ9, whereas Virtuoso requires 46 and Jena 60secs. Most notably, in LQ14 DB2RDF requires 4.6secs while Virtuoso requires 53secs and Jena 94.1secs. For the sub-second queries, DB2RDF is slightly slower than the other systems, but the difference is *negligible* at this scale. So, for LQ1, DB2RDF requires 5ms, while Virtuoso requires 1.8ms and Jena 2.1ms. Similarly, for LQ3 DB2RDF requires 3.4ms while Virtuoso takes 1.8ms and Jena 2.0ms.

The situation is similar in the PRBench case. Figure 17 shows the evaluation time of 4 long-running queries. Consistently, DB2RDF outperforms all other systems. For example, for PQ10 DB2RDF takes 3ms, while Jena requires 27 seconds and Virtuoso requires 39 seconds! For each of the other three queries, DB2RDF takes approx 4.8secs while Jena requires a minimum of 32 and Vir-

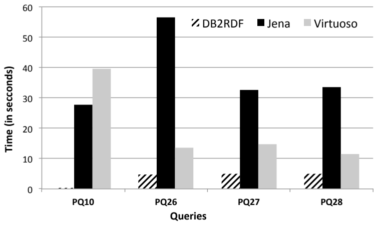


Figure 17: PRBench sample of long-running queries

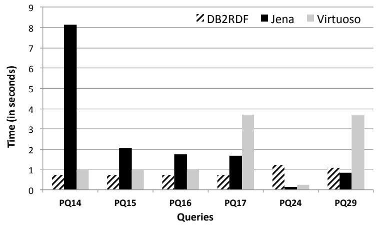


Figure 18: PRBench sample of medium-running queries

tuoso a minimum of 11secs. Figure 18 shows that the situation is similar for medium-running queries where DB2RDF consistently outperforms the competition.

5. RELATED WORK

Abadi et al. [2] propose a predicate-oriented storage for efficient RDF management which uses column stores technology and avoids many of the self-join operations in the final SQL. A recently proposed index structure for RDF called GRIN [20] uses a grouping technique to determine a small subset of the RDF database that contains the answers to a query and can be used independently of the underlying RDF representation. Stocker et al. [17] and Hartig and Heese [8] propose techniques for algebraic re-writing of SPARQL queries to help the query engine devise a better query plan (we already commented about the limitations of these and the following technique in previous sections). Madulo et al. [11] describe a technique to estimate the selectivity of a triple pattern by gathering frequency statistics of the subject, predicate and object and then assuming probabilistic independence between their distributions.

Chen et al. [4] improve the performance of relational-based XML engines. As in our work, the authors store XML documents in a single wide and sparse table. Unlike our work, the management of null values is shifted to the relational engine, and spills are not handled, *i.e.,* it is not clear the work handles the case when an XML element is too large to fit in a record.

Weiss et al. [22] propose an *in-memory* store based on six extended indexes for RDF triples. While the authors argue the benefits of of this approach for query processing, there are important scalability concerns: memory requirements scale linearly with data size and for 6M triples (the largest dataset evaluated) the system requires 8GB of memory. Accordingly, for, say, the 100M LUBM dataset they would require approximately 120GB of memory. Our disk-based store scales without such requirements.

Huang et al. [10] focus on storage and query execution for clustered RDF databases with datasets distributed using graph partitioning algorithms and queries split into chunks that can be executed in parallel. Our work focuses on a centralized setting.

6. CONCLUSIONS

In this paper, we introduced a novel representation and querying mechanism for RDF data on top of relational databases. We introduced DB2RDF, an innovative relational schema that deals with RDF sparsity and schema variability. We showed that DB2RDF has additional benefits during query processing including the reduction of join operations for star queries. We also introduced an innovative SPARQL query optimization technique and novel SPARQL-to-SQL translation techniques, and we showed that these outperform existing RDF stores on standard datasets and query workloads.

For future work, we are preparing a study on insertion, bulk load and update performance and we are planning to extend our system to support the SPARQL 1.1 standard (including property paths). We are also planning to support inferencing, a topic we briefly discussed during the presentation of the LUBM benchmark.

7. REFERENCES

[1] 4store - scalable RDF storage.
http://4store.org/http://4store.org/.

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[3] Best Buy jump starts data web marketing.
http://www.chiefmartec.com/2009/12/best-buy-jump-starts-data-web-marketing.html.

[4] L. J. Chen, P. A. Bernstein, P. Carlin, D. Filipovic, M. Rys, N. Shamgunov, J. F. Terwilliger, M. Todic, S. Tomasevic, and D. Tomic. Mapping XML to a Wide Sparse Table. In *ICDE*, pages 630–641, 2012.

[5] DBpedia dataset. http://dbpedia.org.

[6] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*, 2011.

[7] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2–3):158–182, 2005.

[8] O. Hartig and R. Heese. The SPARQL Query Graph Model for Query Optimization. pages 564–578. 2007.

[9] O. Hassanzadeh, A. Kementsietsidis, and Y. Velegrakis. Data management issues on the semantic web. In *ICDE*, pages 1204–1206, 2012.

[10] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[11] A. Maduko, K. Anyanwu, A. Sheth, and P. Schliekelman. Estimating the cardinality of rdf graph patterns. In *WWW*, pages 1233–1234, 2007.

[12] M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo. DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In *ISWC 2011*, 2011.

[13] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, Feb. 2010.

[14] Resource Description Framework (RDF).
http://www.w3.org/RDF/.

[15] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. *CoRR*, abs/0806.4627, 2008.

[16] SPARQL Query Language for RDF.
http://www.w3.org/TR/rdf-sparql-query/.

[17] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, pages 595–604, 2008.

[18] The New York Times Linked Open Data.
http://data.nytimes.com/.

[19] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. Heuristics-based query optimisation for SPARQL. In *EDBT*, pages 324–335, 2012.

[20] O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A Graph-based RDF Index. In *AAAI*, pages 1465–1470, 2007.

[21] Virtuoso Open-Source Edition.
http://virtuoso.openlinksw.com/wiki/main/main/.

[22] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[23] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Semantic Web and Databases Workshop*, pages 131–150, 2003.