An Extensible Framework for Query Optimization on TripleT-Based RDF Stores

Bart G. J. Wolff
Eindhoven University of Technology
b.g.j.wolff@alumnus.tue.nl

George H. L. Fletcher
Eindhoven University of Technology
g.h.l.fletcher@tue.nl

James J. Lu
Emory University
jlu@emory.edu

ABSTRACT

The RDF data model is a key technology in the Linked Data vision. Given its graph structure, even relatively simple RDF queries often involve a large number of joins. Join evaluation poses a significant performance challenge on all state-of-the-art RDF engines. TripleT is a novel RDF index data structure, demonstrated to be competitive with the current state-of-the-art for join processing. Query optimization on TripleT, however, has not been systematically studied up to this point. In this paper we investigate how the use of *(i)* heuristics and *(ii)* data statistics can contribute towards a more intelligent way of generating query plans over TripleT-based RDF stores. We propose a generic framework for query optimization, and show through an extensive empirical study that our framework consistently produces efficient query evaluation plans.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query processing

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

RDF, SPARQL, TripleT, indexing, query processing

1. INTRODUCTION

Motivation. The goal of the Linked Data vision is to create a global "web of data": an infrastructure for machine-readable semantics for data on the web [9]. This vision aims to make data from a wide variety of sources available under the standardized RDF data model, allowing for this data to be shared across different domains using web standards.

As adoption of the linked data vision grows, data stores have to be able to deal with increasingly large datasets. This poses a scalability problem, both for storage and indexing, as well as for query evaluation. By its triple-centric graph-like nature, even the most basic RDF queries involve a large number of (self-)joins, which pose a significant performance challenge on state-of-the-art RDF database engines. At present, real-world RDF datasets can involve hundreds of millions or even billions of triples, making it challenging to offer interactive query response time.

When compared to relational database technology, RDF stores are a relatively new concept. A number of RDF stores exist, one of them being the Three-way Triple Tree data structure (TripleT) [6], which features a value-based, role-free indexing scheme, unique among the current state-of-the-art. Research has shown this approach to be competitive with, and often at an advantage to, alternative indexing schemes, in terms of both storage and query evaluation costs [6]. However, query optimization on TripleT-based RDF stores has not been systematically studied up to this point.

Our contributions. In this paper, we present our experiences and results of a comprehensive investigation of query optimization on TripleT [18]. In particular, we study how the use of *(i)* heuristics and *(ii)* dataset statistics can contribute towards a more effective generation of query plans, minimizing query execution time over the TripleT RDF store. Our aim here is to understand the effectiveness of various parts of the heuristics that influence query plan generation. These query plans are tailored to (and evaluated on) our implementation of the TripleT store, which we also describe in this paper.

The novelties of our work include an extensible generic rule-based framework for query optimization over TripleT, and an extensive empirical study into the effectiveness of proposed rules in generating optimized query plans. Furthermore, the complete experimental framework, including both disk-based storage and the query processing pipeline, is available as open-source code for further study.[1]

Our proposed optimization framework, together with a few key heuristics rules, is able to consistently produce efficient query plans for a wide variety of query types and datasets. In comparing heuristics-based and statistics-based rules, our aim was to understand the benefit offered by the use of statistics. Our study shows that not only do rules using statistics in general offer little performance improvements compared to heuristics-only rules, but also that a purely heuristics-based approach may exhibit an order of magnitude reduction in evaluation costs in certain situa-

[1] https://github.com/b-w/TripleT

tions. These observations support those of Tsialiamanis et al. in their study of heuristics-based optimization of RDF queries [17].

2. BACKGROUND

Definitions. We present the basics of data and queries. Further details can be found in [1, 18]. Let $\mathbb{U}$ be a set of URIs and $\mathbb{L}$ be a set of literals, such that $\mathbb{U} \cap \mathbb{L} = \emptyset$. Then we define an *RDF triple* as an element $(s, p, o) \in \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$. We define an *RDF dataset* (or, alternatively, an *RDF graph*), denoted $\mathbb{T}$, as a set of $n \geq 0$ RDF triples: $\mathbb{T} = \{t_1, t_2, \cdots, t_n\}$.

At the core of many RDF query languages such as SPARQL lies the concept of Basic Graph Patterns (BGPs) [1]. A BGP is a conjunction of Simple Access Patterns (SAPs), where each SAP is a triple consisting of some combination of fixed values (atoms) and unfixed values (variables). Formally, let $\mathbb{A} = \mathbb{U} \cup \mathbb{L}$ be a set of atoms, and let $\mathbb{V}$ be a set of variables, such that $\mathbb{A} \cap \mathbb{V} = \emptyset$. Then we define an *SAP* as a triple $S = (s, p, o) \in (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{A} \cup \mathbb{V})$. We then define a *BGP* as a conjunction of SAPs: $P = S_1 \wedge S_2 \wedge \cdots \wedge S_n$, for some $n \geq 0$. Equivalently, P may be regarded as the set $\{S_1, S_2, \ldots, S_n\}$.

A *binding* for BGP P is a function B from the variables occurring in P to the set of atoms $\mathbb{A}$. We define the *application* of binding B to P, denoted $B(P)$, as the set of triples resulting from replacing every occurrence of every variable v in P with $B(v)$. Finally, the result of querying graph $\mathbb{T}$ with P, denoted $P(\mathbb{T})$, is the set of all bindings $\mathbb{B}$ such that for each $B \in \mathbb{B}$ it holds that $B(P) \subseteq \mathbb{T}$.

We indicate variables with the prefix '?.' As a small example, the BGP $P = (jan, knows, ?p) \wedge (?p, fanOf, mozart)$ on graph

$$\mathbb{T} = \{(jan, knows, sue), (jan, knows, tim), (sue, fanOf, mozart)\}$$

evaluates to $P(\mathbb{T}) = \{\langle ?p : sue \rangle\}$.

Related work. Numerous RDF stores and indexes have been developed in recent years. Notable examples include Virtuoso [5], RDF-3X [15], and Sesame.[2] We refer the reader to Luo et al. [12] for a thorough survey of storage and indexing solutions for massive RDF datasets.

The study of query optimization is as old as the study of database systems. On the topic of RDF, Neumann and Weikum [14, 15] address some of the scalability problems that arise when processing join queries on very large RDF graphs. Optimizations for BGPs using statistics for selectivity estimation are discussed by Stocker et al. [16], while Tsialiamanis et al. present a number of heuristics for BGP static analysis and optimization [17]. Various studies have been made on techniques for selectivity and cardinality estimation using precomputed information over RDF datasets [7, 10, 13, 15].

TripleT was originally proposed by Fletcher and Beck [6]. Value-based indexing for join processing was also shown to be effective in the context of relational and complex-object databases (e.g., [2, 3, 4]). Some prior work exists featuring TripleT. The performance of different join algorithms on the TripleT index was investigated by Li [11]. An extension of TripleT was used by Haffmans and Fletcher [8] as physical representation of data used for a proposed RDFS entailment algorithm, where it was shown to be good candidate for RDFS data storage.

3. A THREE-WAY TRIPLE TREE

The primary novelty of the TripleT index is that it is built over the individual atoms in a dataset, rather than over complete triple patterns. TripleT uses a number of *buckets* that store the actual triples in the dataset. Each bucket stores all the (s, p, o) triples in the dataset, ordered on some permutation of $\{s, p, o\}$. For instance, an SOP-bucket would (conceptually) store the triples sorted first on *subject*, then on *object*, and lastly on *predicate*. The possible bucket orderings are thus SPO, SOP, PSO, POS, OSP, and OPS, though in our implementation we limit ourselves to using SOP-, PSO-, and OSP-buckets only. The remaining permutations, SPO, POS, and OPS, are symmetrical and are not considered in our investigation. Of important note is that each bucket does not contain the triple part (s, p, or o) that corresponds to its primary sort order. This information is implied by the index and does not need to be repeated.

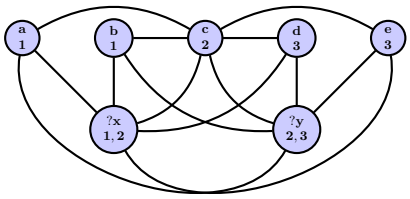
There is one entry in the index for each unique atom in the dataset. This entry contains a number of pointers to triple ranges in each of the bucket files. For instance, the index entry for an atom a might contain a pointer to range $[x \cdots y]$ in the SOP-bucket, meaning that in this bucket, which is sorted on *subject*, triples from position x to position y contain the value a in their *subject* position. Similarly, the same entry might contain pointers to triple ranges in the PSO- and OSP-buckets that contain a in the *predicate*- and *object* positions, respectively.

The index supports retrieval of bindings matching a single SAP. The sort ordering of a bucket determines how suitable it is for retrieving triples matching a particular SAP. An SOP-bucket, for example, would be well suited for retrieving triples matching $(a, ?x, ?y)$, but would be inefficient at retrieving triples matching $(?x, a, ?y)$. For the latter case the bucket ordering implies the entire bucket needs to be read in order to find all possible matches, while for the former case the index entry for a directly points to the range in the bucket where any matches must be contained.

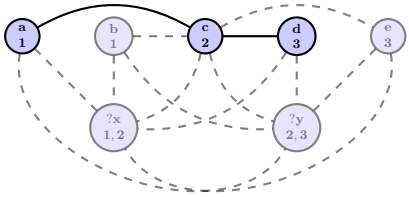
Implementation details. Our work is based on our own open-source implementation of the TripleT RDF store [18]; we briefly highlight salient features here and refer the reader to the full report for further details and design rationale. A single TripleT database is stored on disk across eight different physical files. We use a dictionary to translate between "friendly" representations and internal representations of atoms. This dictionary is stored in two BerkeleyDB[3] hash databases. The TripleT index is stored in a single BerkeleyDB hash database. There is one entry for each unique atom in the dataset. Each bucket belonging to a TripleT database is stored in its own separate file. There are three buckets for each database. The bucket files themselves are flat binary files containing sequences of triples. Each bucket contains all triples belonging to the dataset, although the files themselves contain only the parts of each triple that are not already present in the index. The statistics of a TripleT database are stored in a BerkeleyDB hash database. The

[2] `http://www.openrdf.org`

[3] `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb`



(a) The atom collapse C_{P_1}



(b) A join graph J_{P_1}

Figure 1: Example graphs for the BGP $P_1 = (a, b, ?x)_1 \wedge (?x, c, ?y)_2 \wedge (?y, d, e)_3$

statistical database contains information for estimating output sizes for single SAPs or joins between two SAPs, as well as some summarizing statistics [18].

4. QUERY OPTIMIZATION

Our framework for generating optimized query plans for BGPs over the TripleT index consists of a generic algorithm in which a number of decision points can be manipulated by a given set of rules. In this section, P is defined as a BGP, consisting of k SAPs $(s_1, p_1, o_1), \cdots, (s_k, p_k, o_k)$, denoted $S_1, \cdots, S_k$, resp.

4.1 Atom collapses

We define the *atom collapse* C_P of P as the undirected edge-labeled graph with the atoms and variables of P as nodes, and edges to indicate there is a shared variable between the SAPs associated with nodes.

Formally, the set of nodes consists of *atom*-nodes and *variable*-nodes. For each SAP $S_i \in P$ we have an *atom*-node $(a, \{S_i\})$ for each unique atom $a \in S_i$. We have a *variable*-node (v, P_v) for each unique variable $v \in P$ with $P_v \subseteq P$ being the set of SAPs which contain v. For the special case of SAPs that do not contain any atoms, we have a special atom node $(a_0, \{S_i\})$, where a_0 is a nil-atom.

Let (x, X) and (y, Y) be nodes in the collapse graph. In the set of edges we have an undirected edge $(x, X) - (y, Y)$ with label L if and only if there exists some variable v such that $v \in S_X, S_X \in X$ and $v \in S_Y, S_Y \in Y$ and $S_X \neq S_Y$. Label L consists of a set of (S_i, S_j, v, p_i, p_j) tuples, where there are tuples for every variable v such that $v \in S_i, S_i \in X$ and $v \in S_j, S_j \in Y$ with $S_i \neq S_j$ and with p_i and p_j denoting the positions (s, p, or o) that variable v has in S_i and S_j respectively. Note that one variable can occur in multiple tuples in one label, as long as each tuple as a whole is unique within L.

As an example, Figure 1(a) shows the atom collapse for the BGP $P_1 = (a, b, ?x)_1 \wedge (?x, c, ?y)_2 \wedge (?y, d, e)_3$. For ease of reference, we have numbered the SAPs. Here we have an edge between $(e, \{S_3\})$ and $(?x, \{S_1, S_2\})$, due to the shared variable $?y$ of S_3 and S_2, but no edge between $(e, \{S_3\})$ and $(a, \{S_1\})$, since S_3 and S_1 do not share a variable.

4.2 Join graphs

We define a *join graph* J_P of P as a subgraph of atom collapse C_P, with the nodes from J_P being a subset of the atom-nodes from C_P such that for each SAP $S_i \in P$ there is exactly one node $(a, \{S_i\})$ in J_P, and the edges from J_P being the same as those from C_P. The nodes from the join graph are known as *seed nodes* as they represent the physical access path for each of the SAPs, which is the TripleT bucket used for retrieving them.

Formally, the set of nodes in J_P is defined as

$$NODES(J_P) \subseteq \{(x, X) \in NODES(C_P) \mid x \text{ is an atom}\}$$

such that $\forall S_i \in P.(\exists!(x, X) \in NODES(J_P).(X = \{S_i\}))$. The set of edges is defined as

$$EDGES(J_P) = \{((x, X) - (y, Y) : L) \in EDGES(C_P) \mid (x, X), (y, Y) \in NODES(J_P)\}.$$

Note that P can have multiple distinct, valid join graphs. Figure 1(b) shows a possible join graph for P_1.

4.3 Decision points

The goal of the optimizer is to generate a query plan over the TripleT engine, where a *query plan* is a tree consisting of physical operators as internal nodes and index bucket scans as leaves, for evaluating P. Our optimization framework takes the BGP P as input, first computes its atom collapse C_P, then a join graph J_P, and lastly produces a query plan Q_P for J_P.

The computation features four distinct *decision points*, and we follow a rule-based approach for dealing with them. For computing the join graph there is one such point: (1) deciding which seed node to select from the collapse graph C_P of P. The computation of J_P proceeds by selecting seeds until all SAPs of P are accounted for. For computing the query plan Q_P from J_P there are three: (2) deciding which join edge to select from J_P; (3) deciding which join type to apply for the selected join edge; and, (4) deciding what scan to select for a given SAP. All four decision points are resolved by a number of configurable rules that are separate from the rest of the algorithm.

The rules. Decision points 1 and 4 are identical (both involve selecting a seed for an SAP) and can be resolved by two possible rules. Rule **seed-1** (S1) selects one preferred seed for each distinct SAP in the input set based on the positions of the atoms in the SAP, following the ordering $s \succ o \succ p$. The intuition here is that subjects are more selective than objects, which in turn are more selective than predicates. Rule **seed-2** (S2) does the same but prioritizes the atoms in the SAP according to their selectivity as indicated by dataset statistics.

Decision point 2 has the greatest influence on query response time, as it determines the order of joins in the query plan. It is resolved by five rules. Rule **join-1** (J1) aims to prioritize those joins for which it is possible to do a merge join, which is intuitively cheaper to perform given

the TripleT index organization. Rule **join-2** (J2) prioritizes joins involving the most selective SAPs, where selectivity is determined through the ordering $(s,p,o) \succ (s,?,o) \succ (s,p,?) \succ (?,p,o) \succ (s,?,?) \succ (?,?,o) \succ (?,p,?) \succ (?,?,?)$. Here, s,p,o denote arbitrary atoms and ? denotes an arbitrary variable; and, $S \succ T$ indicates pattern S is more selective than pattern T. Rule **join-3** (J3) aims to prioritize joins between two SAPs that have the most selective positioning of join variables, following the ordering $s \bowtie p \succ o \bowtie p \succ s \bowtie o \succ s \bowtie s \succ o \bowtie o \succ p \bowtie p$. Rule **join-4** (J4) prioritizes joins between SAPs which feature a literal value (e.g. "Sue") in one of its positions, over those featuring only URIs (e.g. "http://example.org/Sue"). The intuition behind rules J2-J4 generalizes our intuition behind S1. Rule **join-5** (J5) prioritizes joins between SAPs which, according to the statistics database over the dataset, produce the smallest intermediate result sets.

Decision point 3 is resolved by a fixed heuristic: whenever it is possible to do a merge join (i.e. the left- and right input sets involved in the join are both sorted on their shared join variables), we do so; if not, we perform a hash join instead.

The rules for resolving decision points 1, 2, and 4 can be used in any configuration (i.e. which rules are and are not used, and in which order are they applied). Hence at each decision point there is a variable, ordered list of rules R which act as filters and which are applied in sequence on the set of options in order to arrive at a final choice. Each rule $r \in R$ reduces the set of options I to a set of options $I' \subseteq I$ through filter step $I \xrightarrow{r} I'$. Any items in I' are then said to be equivalent under r. Similarly, an ordered list of rules R performs filtering step $I \xrightarrow{R} I'$, with any items remaining in I' being called equivalent under R.

5. EXPERIMENTAL STUDY

The goal of our experiments is to gather evidence relevant to answering the following questions:

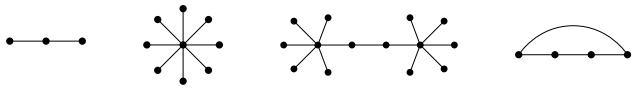
1. How effective is each individual rule for generating optimized query plans?
2. How effective are combinations of rules for generating optimized query plans?
3. Does the order in which rules are applied matter?
4. What is the impact of using statistics?
5. How do our optimization techniques perform under different types of queries?
6. How do our optimization techniques perform under different kinds of datasets?

These questions can be divided into four sections: *(a)* the value of rules, *(b)* the value of statistics, *(c)* the difference between queries, and *(d)* the difference between datasets.

The value of rules. As discussed in Section 4.3, our optimization techniques make use of a number of different rules, which are applied in a certain sequence when we arrive at a decision point where a choice needs to be made. Most of them work based on some heuristic. One would not expect each rule to be just as effective as the next; in fact, such would be a highly surprising outcome. Instead, one would expect there to be noticeable differences in the effectiveness of individual rules. One would also expect that certain combinations of rules will prove to be highly effective, more so than what the sum of the parts might suggest. The order in which rules are applied at a decision point would be expected to matter to a certain degree but be less important than which rules are and are not used.

The value of statistics. Although we have described only two rules in Section 4.3 which make use of statistics, their purpose is the same as that of all of our heuristics-based rules: to minimize intermediate result sizes produced during query plan execution. Of course, the use of these statistics-based rules comes at a cost: a full statistics database needs to be computed and maintained over the dataset.

The difference between query types. There are several different types of queries we use in our experiments. As our datasets are essentially graphs and our queries are graph patterns, it's easy to visualize them as such. In Figure 2 the four common query shapes that our queries are based around are shown, where a query's SAPs are represented by nodes which are connected if they share a variable.



(a) Chain query (b) Star query (c) Star-chain query (d) Loop query

Figure 2: Different query shapes we study

Aside from their **shape**, other variables we study are query **size** (in number of SAPs), and query **selectivity**. The influence of query size on execution time is difficult to predict. On one hand, more SAPs means more joins; on the other, more SAPs can also mean higher selectivity which can be exploited by the plan generator. As for query selectivity, a query which features more atoms in more selective positions in its SAPs generally produces a smaller result set. Again, selective SAPs in a query can be favorably exploited by the plan generator.

The collection of concrete queries used in our experiments – covering the full range of combinations of shape, size, and selectivity – is detailed in Wolff [18].

The difference between datasets. The test data we have used comes from three different sources, covering both real and synthetic data: DBpedia,[4] SP^{2}Bench,[5] and UniProt.[6] From each source we have obtained three different datasets: one 100.000 (100K) triples dataset, one 1.000.000 (1M) triples dataset, and one 10.000.000 (10M) triples dataset. For all datasets, the 100K set is a strict subset of the 1M set, which in turn is a strict subset of the 10M set.

Plan of study. In our experiments we primarily compare different rule sets to each other. Hence, the composition of the rule set is the main variable in each experimental run. We define a *run* in our experiment as testing one particular rule set on every dataset using every available query. Here, *testing* some rule set x on dataset y using query z is com-

[4] http://dbpedia.org/

[5] http://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/

[6] http://www.uniprot.org/

Table 1: Results overview on 1M datasets: execution time (ms)

	SP²Bench 1M					UniProt 1M				
	Chain	Star	Star-chain	Selective	Non-selective	Chain	Star	Star-chain	Selective	Non-selective
Run A-3	9975,5	6806,2	5070,5	2376,5	12191,6	1151,9	24328,1	25884,0	17512,9	16729,7
Run A-4	18121,6	6293,4	198520,0	55936,2	92687,1	1194,7	39200,0	2173,3	1472,3	26906,4
Run D-2	12173,2	6876,5	5195,6	3018,5	13145,0	1158,0	28490,1	3620,1	4010,5	18168,2

prised of: opening the TripleT database for dataset y; telling the query plan generator to use rule set x; feeding query z to the database; enumerating and immediately discarding the query results; closing the database. Each run is tested five times, each time "cold", i.e. without preserving any caches between tests, with average costs reported.

The runs we have performed are detailed in Table 2, where a number indicates that a particular rule was used in that run, the number itself indicating the order (a lower number denoting a higher priority). Each of the runs has a particular purpose: the **A-runs** are designed to test the heuristics rules against the statistics rules; the **B-runs** aim to get a sense of the value of the individual heuristics rules; the **C-runs** focus on the ordering of rules; the **D-runs** are used to test different subset combinations of rules.

Table 2: Runs and their rule sets

	Rules						
Runs	S1	S2	J1	J2	J3	J4	J5
A-1	1		1	2	3	4	
A-2		1					1
A-3	1	2	1	2	3	4	5
A-4	2	1	2	3	4	5	1
B-1	1			1	2	3	
B-2	1		1		2	3	
B-3	1		1	2		3	
B-4	1		1	2	3		
C-1	1		4	3	2	1	
C-2	1		3	2	1	4	
C-3	1		2	1	4	3	
D-1		1	1	3			2
D-2	1		2	1			
D-3		1		1	2		3

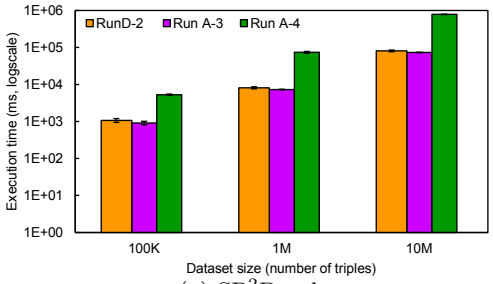
5.1 Empirical results

In the interest of space, we highlight only a few of our main observations here and refer the reader to Wolff [18] for a detailed presentation and analysis of all results.

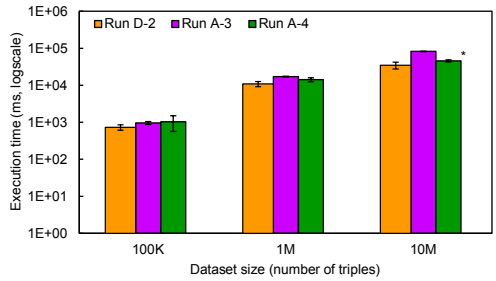
The A-runs. These runs were designed to test the performance of the heuristics-based rules (runs A-1 and A-3) against the statistics-based rules (runs A-2 and A-4).

We focus our discussion on A-3 and A-4 as illustrations of these two groups. Table 1 presents results for the SP²Bench and UniProt 1M datasets. We visualize results over all dataset sizes in Figure 3, where we plot the average query execution time over all queries, along with the average standard deviation. These results confirm that the heuristics-based A-3 rule set offers better performance overall.

For the SP²Bench datasets, the heuristics-based A-3 is most effective, with execution time of A-4 reaching up to an order of magnitude higher. The DBPedia datasets showed performance similar to that of SP²Bench. On UniProt the differences are less pronounced, though we note that A-4 failed to scale up to the 10M set for the star-chain query.



(a) SP²Bench



(b) UniProt

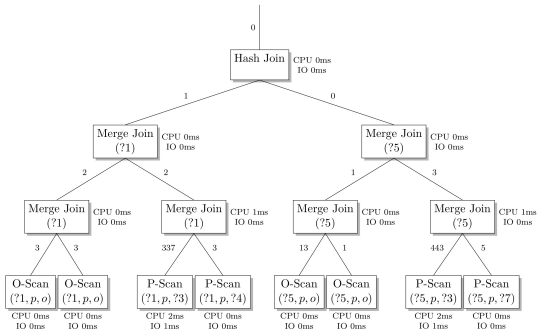
Figure 3: Results overview of runs A-3, A-4, and D-2

Here, a combination of primarily heuristics rules with statistics as back-up worked best, as seen with A-3.

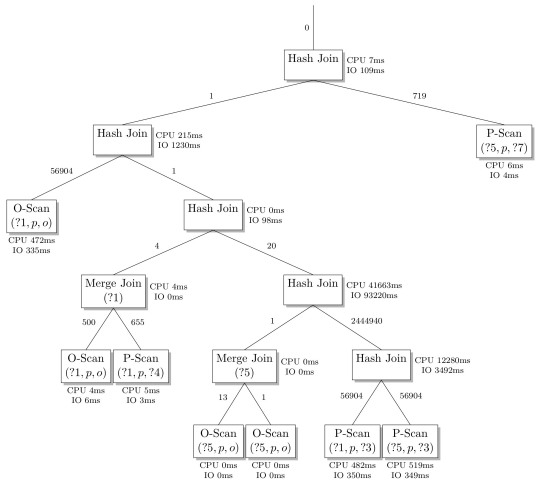
On closer inspection, the general cause for A-4's performance is that its plans feature more hash joins than those produced by A-3. Performing a hash join can result in a significant amount of intermediate result materialization, whereas this is not the case with the merge join because it can take advantage of the fact that the input streams are guaranteed to be sorted. This is illustrated by the actual plans generated by A-3 and A-4 as presented in Figure 4, for the following small selective star-chain SP²Bench query:

(?1, http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
http://localhost/vocabulary/bench/Article),
(?1, http://purl.org/dc/elements/1.1/creator,
http://localhost/persons/Paul_Erdoes),
(?1, http://swrc.ontoware.org/ontology#journal, ?3)
(?1, http://purl.org/dc/terms/references, ?4),
(?5, http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
http://localhost/vocabulary/bench/Article),
(?5, http://purl.org/dc/elements/1.1/creator, Dell_Kosel),
(?5, http://swrc.ontoware.org/ontology#journal, ?3),
(?5, http://purl.org/dc/terms/references, ?7)

The B- and C-runs. These runs were designed to get a sense of the value for each individual heuristics rule and to measure the importance of rule ordering, respectively. In



(a) Run A-3



(b) Run A-4

Figure 4: Plans generated for a small selective star-chain query on the SP^{2}Bench 1M dataset

short, we observed in these experiments that the J1 rule is the single most important heuristics rule: configurations that gave it a lower priority often failed to scale up to the 10M datasets. Aside from that there appeared to be no clear winner in individual rules or ordering of the rules.

The D-runs. In these runs we look at configurations which use a limited subset of rules. These experiments showed that minimal rule sets are quite stable and effective in performance. As a point of comparison with A-3 and A-4, we present the results for D-2 in Table 1 and Figure 3. Here we see that even this very limited rule set is always competitive with both A-3 and A-4. The D-2 run appears to provide the best, consistent performance over all three datasets, and preferring the J2 rule (selectivity) over the J1 rule (merge-joins) is one of the few configurations to perform well on the UniProt star- and star-chain queries, which proved to be some of the most difficult queries we have tested. This performance by the D-2 run is somewhat surprising, as this configuration consists of only three heuristics rules (one seed rule, two join rules), and does not use statistics at all.

5.2 Discussion

We have seen through an extensive empirical evaluation how the value of individual rules used by the plan generator can vary greatly. Especially the merge join prioritization rule, which is given preference in A-3, appears to be invaluable for the generation of efficient query plans. As a heuristic, there are of course practical scenarios which violate this good behavior of merge join prioritization. In our experiments, we experienced this only in the case of UniProt star-chain queries. Indeed, here the performance generally improved when this rule was given a lower priority, as in D-2, where the selectivity prioritization rule proved to have the largest positive impact. The benefits of the three remaining heuristics rules are similar. In particular, in their absence, the impact on query response time is roughly the same.

In general, we have observed that the impact of statistics and the statistical prioritization join rule is measurable but limited. When used alone or as the primary join rule, the statistics rule produces query plans significantly worse than those produced by the heuristics rules, as evidenced by run A-4. This suggests that the value of statistics rules is found mostly in a supporting role.

In summary, we recommend the D-2 rule configuration for general use, as it is a purely heuristic and minimal approach which delivers excellent results across the board. Overall, our findings corroborate results obtained by Tsialiamanis et al. [17], where a heuristics-based planner for SPARQL queries is shown to be competitive with the cost-based approach taken in the state of the art RDF-3X store [15].

6. CONCLUDING REMARKS

In this paper we have presented results of a study of query optimization on TripleT-based RDF stores. We have proposed a query optimization framework that takes the shape of a generic, rule-based algorithm. We also proposed a number of heuristic and statistical rules for use by this algorithm.

We have evaluated this framework in an extensive series of experiments. These experiments have shown that a small number of relatively simple heuristics can consistently produce efficient evaluation plans for a wide variety of queries and datasets. We have also seen that while statistics do add value, the value is minimal, and not within reasonable proportion to the costs involved in constructing and maintaining statistical data structures over massive graphs.

A number of interesting avenues for future work remain open. A study of runtime optimization strategies in our framework, such as sideways information passing [14], and further sophisticated join-ordering [7] strategies are both naturally rich areas for exploration. We have also encountered various challenges with using statistics for query optimization. Additional work in this area would be interesting, and may yet help our optimization framework produce even more efficient query plans.

Acknowledgments. We thank Antonio Badia, Paul De Bra, and Herman Haverkort for their helpful comments.

References

[1] M. Arenas, C. Gutierrez, and J. Pérez. Foundations of RDF databases. In *Reasoning Web*, pages 158–204, Brixen-Bressanone, 2009.

[2] B. C. Desai. Performance of a composite attribute and join index. *IEEE TSE*, 15(2):142–152, 1989.

[3] A. Deshpande and D. Van Gucht. A storage structure for nested relational databases. In *Nested Relations and Complex Objects, LNCS 361*, pages 69–83. Springer, 1987.

[4] A. Deshpande and D. Van Gucht. An implementation for nested relational databases. In *VLDB*, pages 76–87, Los Angeles, 1988.

[5] O. Erling and I. Mikhailov. RDF support in the Virtuoso DBMS. In T. Pellegrini et al, editor, *Networked Knowledge - Networked Media*, pages 7–24. Springer, 2009.

[6] G. H. L. Fletcher and P. W. Beck. Scalable indexing of RDF graphs for efficient join processing. In *CIKM*, pages 1513–1516, Hong Kong, 2009.

[7] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*, pages 439–450, Athens, Greece, 2014.

[8] W. J. Haffmans and G. H. L. Fletcher. Efficient RDFS entailment in external memory. In *SWWS*, pages 464–473, 2011.

[9] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[10] H. Huang and C. Liu. Estimating selectivity for joined RDF triple patterns. In *CIKM*, pages 1435–1444, Glasgow, 2011.

[11] K. Li. Cost analysis of joins in RDF query processing using the TripleT index. Master's thesis, Emory University, 2009.

[12] Y. Luo, F. Picalausa, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. Storing and indexing massive RDF datasets. In R. De Virgilio et al, editor, *Semantic Search over the Web*, pages 31–60. Springer, 2012.

[13] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984–994, Hannover, Germany, 2011.

[14] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, pages 627–640, Providence, Rhode Island, USA, 2009.

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

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

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

[18] B. G. J. Wolff. A framework for query optimization on value-based RDF indexes. Master's thesis, Eindhoven University of Technology, 2013. http://alexandria.tue.nl/extra1/afstversl/wsk-i/wolff2013.pdf.