OptARQ: A SPARQL Optimization Approach based on Triple Pattern Selectivity Estimation

Abraham Bernstein, Christoph Kiefer, Markus Stocker
Department of Informatics
University of Zurich
{bernstein,kiefer,stocker}@ifi.unizh.ch

Technical Report No. ifi-2007.03

March 2, 2007

Abstract Query engines for ontological data based on graph models mostly execute user queries without considering any optimization. Especially for large ontologies, optimization techniques are required to ensure that query results are delivered within reasonable time. OptARQ is a first prototype for SPARQL query optimization based on the concept of triple pattern selectivity estimation. The evaluation we conduct demonstrates how triple pattern reordering according to their selectivity affects the query execution performance.

Paper Type: Technical Report

1 Introduction

Since the advent of System R [1], query optimization has always been a research topic. The techniques introduced by D. Chamberlin and his IBM research group in San José, California, in 1979 are still used in commercial database systems [2]. The concept of selectivity factor described in [1] is still a crucial one. Clearly, selectivity estimation methods evolved from simply formulas to more complex statistical techniques, but the foundations are applicable to modern query languages like SPARQL.

The proposed optimization framework for SPARQL queries mainly focuses on static optimization techniques. By static optimization we mean general rules which may be applied in order to get an optimal query execution plan (QEP). Thus, static optimization is a query rewriting process usually executed after query parsing and syntax checking.

The fundamental aim of the proposed optimization framework is to reduce intermediate result sets of triple patterns. Basically, each implemented optimization rule is intended to fulfill this goal. For example, the triple pattern selectivity estimation enables a pattern ranking according to their intermediate result set sizes.

The paper is structured as follows: Next, we review the general concept of selectivity of a condition, we propose our adaptation to the selectivity of triple patterns and we present our approach to the selectivity estimation of triple patterns. Then, we provide a detailed explanation of the implemented optimization rules. In our evaluation we discuss the usefulness of the approach. Finally, we discuss the limitations of the presented approach. We close with a discussion of related work and some ideas for future work.

2 Selectivity

Selectivity is the most crucial concept on which our optimization model is based. Piatetsky defined the selectivity in [3] as follows.

Definition 1 *Selectivity of a condition E, denoted SEL(E), is the fraction of tuples satisfying this condition.*

For example, SEL(number = 6) is about 0.16 for a typical dice. This definition can be adapted with some small modification to ontological data models. We may modify the definition for SPARQL queries, to

Definition 2 *Selectivity of a triple pattern T, denoted SEL(T), is the fraction of triples satisfying the pattern.*

Selectivity is fundamental because it quantifies the size of intermediate result sets of triple patterns. Thus, the overall goal is to find the ordering of query patterns which minimizes the intermediate result sets for each stage during query execution.

The optimization we focus on is based on static query rewriting rules. In the following we define some general rules for SPARQL query rewriting and we describe the histogram based model for triple pattern selectivity estimation.

2.1 Selectivity Estimation

Selectivity can be calculated either by an exact formula or an estimation which is mostly based on statistics about the underlying data. Because an exact triple pattern selectivity computation basically requires the pattern to be executed, we cannot rely on exact information since this would require as much time to perform the optimization as it is required to execute the query. Thus, we base our optimization model on statistical information about the ontological resources in order to get an estimation of the selectivity for each triple pattern. This allows to rank the patterns according to their estimated selectivity which is expected to reduce the intermediate result set sizes. This may results in considerable performance improvement.

The evaluation (Section 4) shows that the estimation based on statistics is enough precise to reduce the execution performance by orders of magnitude.

2.2 Selectivity Cost Function

We define a cost function that reflects the selectivity estimation and is used to rank triple patterns in increasing order of selectivity. The cost function returns a value between 0 and 1, thus, it is basically a normalization to [0,1] of the estimated selectivity.

We model the overall cost for a triple pattern as follows

$$c(t) = c(s) * c(p) * c(o)$$

where $c(t)$ is the overall cost for a triple pattern t and s, p, o are respectively the subject, predicate and object of t. Thus, the expected execution cost for t, $c(t)$, is modeled as the multiplication of the expected cost for the subject $c(s)$, predicate $c(p)$, and object $c(o)$.

Subject Cost Estimation

The subject of a triple pattern may be either a variable or an IRI. In the case of a variable, we assign the cost 1.0 since we miss information to make a more precise cost estimation. In the case of an IRI the subject matches a resource in the graph model. Thus, the exact number of triples returned by a pattern where the subject is specified by an IRI corresponds to the number of predicates defined for the referenced resource. In order to avoid creating an index containing the exact information for each resource, our statistical model estimates the cost of a pattern where the subject is specified with an IRI by

$$c(s) = \frac{1}{|R|}$$

where $|R|$ is the total number of resources in our ontology. This results in a constant for the selectivity of subjects in the queried ontology which is fairly a rough estimation that holds well only for ontologies containing resources of the same class (e.g., publications) or queries addressing just one class of subject. We will investigate more precise estimations in future work.

Predicate Cost Estimation

The predicate of a triple pattern can be either a variable or an IRI. In the case of a variable, we again assign the cost 1.0 as for the subject since we miss information to make a more precise estimation. In the case of an IRI, it matches each triple which features the IRI as predicate. We estimate the cost for predicate p by

$$c(p) = \frac{|T_p|}{|T|}$$

where $|T_p|$ corresponds to the (exact) number of triples matching predicate p and $|T|$ is the total number of triples. This is the fraction of triples which matches predicate p. Thus, the predicate cost estimation given an IRI is exact. Since the number of distinct predicates defined in ontologies is mostly marginal compared to the number of resources or triples, it is worth to index the exact information for predicates in order to get a precise selectivity.

Object Cost Estimation

The triple pattern object can be either a variable, an IRI or a RDF literal. In the case of a variable, we assign the cost 1.0 as for the subject and the predicate. Otherwise, we extract the estimated selectivity from a histogram. The object values domain for predicates is represented by histograms, more precisely equal-width histograms [3]. As we will depict later on in this chapter, the range of object values is divided into B equal-width histogram classes where the class height corresponds to the number of objects given a predicate that fall into the class. Hence, for each predicate a histogram of the corresponding object value domain is created. We estimate the object cost $c(o)$ by

$$c(o) = \begin{cases} c(p, o_c), & \text{if p is bound;} \\ \sum_{p_i \in \mathbb{P}} c(p_i, o_c), & \text{otherwise.} \end{cases}$$

where $c(p, o_c) = \frac{h_c(p,o_c)}{|T_p|}$, i.e., the frequency of o_c (i.e., the height of the histogram class for o_c) normalized by the number of triples matching p and o_c is the histogram class in which the object o falls into. In the case a predicate is not bound, the histogram of each predicate in the model is considered for the object cost estimation. Please note that histograms represent the object values domain of a specific predicate. Thus, to address a specific histogram the predicate IRI is required.

2.3 Examples

In order to illustrate the cost model described in the previous Section 2.2, we illustrate some examples. They are based on a sample ontology O. First, we perform a process over O to gather the required statistics. O contains 1'317 triples and an average of 11.52 predicates for each resource. Moreover, the predicate RDFS:label[1] appears in 114 triples. O is a publication ontology and the predicate RDFS:label is used to describe the title of publications. The title 'XQuery: A Query Language for XML' [4] falls into a histogram class of height 17.

[1]http://www.w3.org/2000/01/rdf-schema#label

Based on these statistics we are now able to estimate the cost of triple patterns. Table 1 illustrates some examples which show the estimated execution cost depending whether a subject, predicate or object are variable or not.

	t	c(s)	c(p)	c(o)	c(t)
1	?s ?p ?o	1.0	1.0	1.0	1.0
2	:s ?p ?o	0.008747	1.0	1.0	0.008747
3	?s rdfs:label ?o	1.0	0.0865604	1.0	0.0865604
4	:s rdfs:label ?o	0.008747	0.0865604	1.0	0.0007571
5	?s rdfs:label ”XQuery: A Query ...”	1.0	0.0865604	0.0129081	0.0011173
6	:s rdfs:label ”XQuery: A Query ...”	0.008747	0.0865604	0.0129081	0.0000097

Table 1: Triple Pattern Cost Estimation

2.4 Statistical Model

The required statistical information about the underlying graph model is (usually) previously extracted by a pre-process step. There is a special case where the statistics may be extracted during query execution, namely when the unoptimized query execution is expected to be more time consuming than the process which gathers the statistical information. During our evaluation we experienced that this special case may be surprisingly more frequent than expected.

In either case, the implementation of our ARQ optimizer, *OptARQ*, requires a statistical model to estimate the triple pattern cost. We implement an application on top of Jena[2] which creates an ontology model containing all required statistical information. The model is serialized to a RDF/XML representation which can be loaded into a query execution environment.

The statistical model is represented as a graph. It contains a `stat:Statistics` resource with both predicates `stat:avgNrOfPredicates` holding the average number of predicates for any resource and `stat:nrOfTriples` for the number of triples contained in the ontology. Furthermore, the model contains a `stat:Predicates` resources with a RDF sequence[3] which lists all distinct predicates contained in the ontology. Each predicate in the list is described on its part by a RDF resource which defines a predicate `stat:frequency` for the absolute number of occurrences the predicate appears in triples and a predicate `stat:histogram` which is a reference to the histogram representation for the object values of the corresponding predicate. A histogram representation is again a resource which contains a RDF sequence of histogram classes. Each histogram class defines a reference to a RDF resource with a predicate `stat:label` which specifies the histogram class lower bound and a predicate `stat:items` which describes the histogram class height, i.e., the number of elements falling into the class.

3 SPARQL Optimization

This section focuses on SPARQL optimization. We describe the general optimization rules considered in our optimization framework which are used to rewrite SPARQL queries in order to get

[2]http://jena.sourceforge.net

[3]http://www.w3.org/TR/rdf-primer/

Listing 1: Example: Rewrite Filter Variables

```
PREFIX  person:        <http://person/>
PREFIX  publication:   <http://publication/>

SELECT ?person ?firstname ?title
WHERE {
        ?person person:firstname ?firstname .
        ?person person:lastname ?lastname .
        ?person person:age ?age .

        ?publication publication:author ?person .
        ?publication publication:title ?title .

        FILTER (?firstname = "Donald"
                && ?lastname = "Chamberlin" && ?age > 30)
}
```

an optimized QEP. Rules contain a prepare and a transform stage. During prepare stage we gather information about the query. The information is subsequently used during transform stage in order to rewrite the query according to a specific optimization goal. Please note that the execution order of rules is relevant since the estimated triple pattern execution cost may vary after applying rewriting rules (e.g., FILTER rewriting).

3.1 Rewrite Filter Variables

The purpose of this rule is to inspect whether FILTER expressions can be decomposed and variables included in expressions eliminated by substituting the value directly in some triple pattern. There are a couple of issues to consider. First, we can decompose only FILTER expressions that are connected by an AND logical operator. SPARQL triple patterns are joined by a logical AND. Thus, if we substitute a triple pattern variable, we need to make sure that the variables in FILTER expressions are connected by AND. The substitution of variables connected by OR would lead to a different semantic and, thus, to a different result set. Another important remark is that a variable in some triple pattern cannot be simply substituted if the variable is read (referenced) somewhere else in the query (for example in query projection). Thus, we need to make sure that substituted variables occur only once in query. Finally, we need to consider the operator used in FILTER expressions to contrain a variable by a value. There is only one case that a variable can be substituted, namely in the case of an equal operator (=).

Listing 1 shows an example where the optimization rule may be applied. Please mark that some of the issues described above need to be considered. The variables defined in FILTER expression are connected by a logical AND. Hence, they may be decomposed. Furthermore, the variable `?firstname` is listed in projection variables. Thus, we cannot simply rewrite the first triple pattern (Listing 1).

At this point we have two options. We could leave the triple pattern with the variable and filter

Listing 2: Optimized Example: Rewrite Filter Variables

```
PREFIX   person:         <http://person/>
PREFIX   publication:    <http://publication/>

SELECT ?person ?firstname ?title
WHERE {
        ?person person:firstname "Donald" .
        ?person person:lastname "Chamberlin" .
        ?person person:age ?age .
        ?person person:firstname ?firstname .

        ?publication publication:author ?person .
        ?publication publication:title ?title .

        FILTER (?age > 30)
}
```

the variable in FILTER expression. A more optimized alternative is to rewrite the `?firstname` variable for the first triple pattern (Listing 1) and to add the same triple pattern holding the variable as object as last pattern in the basic graph pattern. This way, we very early constrain the intermediate result set by matching only resources with first name 'Donald'. Because the variable `?firstname` is specified in projection we need to add a triple pattern containing the variable as object. Further, we need to pay attention to the variable `?age` defined in FILTER expression. Age is of type integer and the operator used is '$>$'. Thus, we cannot rewrite the `?age` variable specified in the pattern.

The optimized query is displayed in Listing 2. Please note that `?lastname` is the only variable which could be rewritten without any further modification.

3.2 Move Up Filter

The purpose of this rule is to decompose FILTER expressions and execute them as early as possible in query, i.e., after the first basic graph pattern which introduces the corresponding variable. We need to make similar considerations as for the rule described above (Rewrite Filter Variables). In fact, we can decompose FILTER expressions only if the FILTER elements are connected by a logical AND.

As an example, we take the query specified in Listing 2. There is a FILTER expression defined at the bottom of the query. We can move the FILTER expression closer to the `?age` variable defined as object in the pattern. The optimized version of the query is listed in Listing 3. The FILTER move up may constrain the intermediate result set size of the first basic graph pattern matching the `person` resources. This results in a smaller join with publications and thus the execution performance may be optimized.

Listing 3: Optimized Example: Move Up Filter

```
PREFIX  person:        <http://person/>
PREFIX  publication:   <http://publication/>

SELECT ?person ?firstname ?title
WHERE {
        ?person person:firstname "Donald" .
        ?person person:lastname "Chamberlin" .
        ?person person:age ?age .
        ?person person:firstname ?firstname .
        FILTER (?age > 30)

        ?publication publication:author ?person .
        ?publication publication:title ?title
}
```

3.3 Reorder by Selectivity

The purpose of this rule is to reorder triple patterns according to their estimated selectivity. Please refer to the discussion of triple pattern selectivity estimation used in our optimization approach in Section 2.1.

The rule computes the expected execution cost for each triple pattern using our cost function (Section 2.1). This is done in the prepare stage of the rule. During the transform stage triple patterns are reordered according to their costs in increasing order (i.e., increasing selectivity). Thus, triple patterns that potentially yield to smaller intermediate result sets are executed first. As we show in Section 4 this transformation is very fundamental and yields to significant SPARQL optimization.

4 Evaluation

In this section we describe the evaluation approach used to evaluate the SPARQL query execution performance. We illustrate the methods, datasets, query engines, and retrieval tasks used and we present our findings. The evaluation focuses on execution performance of SPARQL queries on a sampled dataset. We show how the performance of retrieval tasks scales for multiple query engines. The evaluation is based on a dataset which fits into main memory. Thus, the results presented in this section focuses on execution performance evaluation of SPARQL query engines with in-memory models. Although the considerations we made in Section 3 about query optimization and the evaluations presented in this chapter are valid also for persistent triple stores, the results and charts presented here are valid for in-memory models only.

We conduct our experiments on a two processor dual core AMD Opteron 270 2.0 GHz server with 4 GB main memory and two 150 GB 7200rpm disks with a 32 bit version of Fedora Core 5 as operating system.

4.1 Query Engines

We evaluate the SPARQL query performance on different query engines, namely ARQ[4], Sesame[5] and KAON2[6]. The optimized ARQ engine, called `OptARQ`, is used as reference for comparison to other engines. ARQ is a query engine for Jena. Sesame is a RDF database with support for RDF-Schema inferencing and querying. Sesame was originally developed by Aduna[7] for an EU research project and is still maintained by Aduna in collaboration with the community and NLnet Foundation[8]. Sesame supports an own query language called SeRQL[9] and provides a SPARQL engine which is developed third party by Ryan Levering[10]. KAON2[11] is an infrastructure for managing OWL-DL, SWRL, and F-Logic ontologies.

The SwetoDblp Dataset

SwetoDblp[12] is a RDF representation of the DBLP[13] publication database and is published by the Large Scale Distributed Information Systems (LSDIS) lab, University of Georgia, USA. SwetoDblp is a spin-off of the Semantic Web Technology Evaluation Ontology (SWETO)[14] and is intended as an infrastructure for testing the scalability of new software. The schema-vocabulary of SwetoDblp aggregates concepts from FOAF[15], Dublin Core[16] and OPUS (specific to the LSDIS library). SwetoDblp contains approximatively 1.3 million resources with a size of 787 MB (November 2006). The considerable size of the ontology allows an extensive query execution performance evaluation.

SwetoDblp Sampling

In order to evaluate the scale performance of retrieval tasks for different query engines, we create a set of samples of the full SwetoDblp ontology. The sampling growth is set to approximatively 10%. Thus, we sample the complete ontology in 10 samples. Table 2 illustrates the sample sizes in mega bytes including the number of triples and resources. Moreover we list the resulting set size in number of resources for or retrieval task. Please note the linear growth of the result sets for each sample. We create samples with linear growth for the returned tuples in order to avoid that matching resources are clustered in a subset of the samples.

Sample (%)	Size (MB)	Triples	Resources	Result Set Size
10	78.7	893’965	79’733	2
20	157.3	1’787’629	159’467	4
30	235.8	2’681’237	239’203	6
40	314.6	3’573’684	318’934	8
50	393.4	4’468’666	398’665	10
60	472.0	5’360’604	478’401	12
70	550.6	6’253’930	558’112	14
80	629.4	7’145’344	637’852	16
90	708.1	8’040’767	717’574	18
100	786.5	8’933’272	797’278	20

Table 2: SwetoDblp Samples: Sample Size, Number of Triples, Number of Resources and Result Set Size for the Retrieval Tasks

4.2 Retrieval Task

We specify a retrieval task for SwetoDblp which reflects a common usage of the ontology. The task (Listing 4) focuses on articles published by a journal during a specific year. It extracts all articles published in 2004 by VLDB journal[17]. An article is described by its title, the researchers that authored the publication, the journal volume and number, the publication year and the number of pages.

4.3 Optimizations

In order to better understand the evaluation, we first describe how our optimization approach rewrites the input query for the retrieval task. The optimized query returned is expected to be optimal according to the statistical selectivity estimation. In addition we shortly describe the optimizations executed by Sesame for the SeRQL query language. SeRQL is a RDF/RDFS query language developed by Aduna[18] as a part of Sesame[19].

[4]http://jena.sourceforge.net/ARQ/
[5]http://www.openrdf.org/
[6]http://kaon2.semanticweb.org/
[7]http://www.aduna-software.com/
[8]http://www.nlnet.nl/
[9]http://www.openrdf.org/doc/sesame/users/ch06.html
[10]http://ryan.levering.name
[11]http://kaon2.semanticweb.org
[12]http://lsdis.cs.uga.edu/projects/semdis/swetodblp/
[13]http://dblp.uni-trier.de/
[14]http://lsdis.cs.uga.edu/projects/semdis/sweto/
[15]http://xmlns.com/foaf/0.1/
[16]http://dublincore.org/
[17]http://www.informatik.uni-trier.de/ ley/db/journals/vldb/index.html
[18]http://www.aduna-software.com/
[19]http://www.openrdf.org/doc/sesame/users/ch06.html

Listing 4: Retrieval Task

```
PREFIX  opus:  <http://lsdis.cs.uga.edu/propono#>
PREFIX  rdfs:  <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?author ?volume ?pages ?number
WHERE {
        ?article opus:year ?year .
        ?article opus:publication_authored_by ?author .
        ?article rdfs:label ?label .
        ?article opus:volume ?volume .
        ?article opus:pages ?pages .
        ?article opus:number ?number .
        ?article opus:journal_name ?journal_name .

        FILTER (?year = 2004 && ?journal_name = "VLDB J.")
}
```

The query of our retrieval task (Listing 4) is optimized by two different optimization rules. First, we rewrite the FILTER expression. During the prepare stage, the optimizer searches for FILTER expressions and checks whether they can be rewritten. We may rewrite a FILTER expression when its elements are connected by a boolean AND operator and they are compared by equal operator (=). Since both variables `?year` and `?journal_name` are not read elsewhere in the query, we can just overwrite both object variables in the corresponding triple patterns. Further, we apply the rule 'Reorder by Selectivity'. During prepare stage of the rule, the optimizer calculates the estimated triple pattern execution costs as a function of the estimated selectivity (please refer to Section 2 for further details). We get a set of [0,1]-values used in transformation stage to reorder the triple patterns. The optimized query is listed in Listing 5. After rewriting the FILTER expression the first triple pattern (line 6) has the smallest selectivity, i.e., lowest cost and is thus placed first. This is reasonable compared to the second triple pattern (line 7) since the articles published by the VLDB journal are expected to be less than the articles published in 2004. The following three patterns (lines 8 - 10) are more difficult to anticipate, but it is reasonable that there are more triples matching the predicate `opus:number` than triples matching the second triple pattern (line 7) since not only the articles published in 2004 but every article will match the third pattern (line 8). Because the ontology not only contains articles but other resources too (e.g., master thesis) it is straightforward that the sixth pattern (line 11) is placed after the pattern with the `opus:volume` predicate (line 10). Only article resources match the `opus:number` predicate whereas each resource matches the `rdfs:label` predicate (i.e., the resource title). Last but not least, we may explain why the seventh pattern (line 12) is placed last. Since an article features only one title but it is often written by one or more authors, the last pattern potentially matches more triples. Thus, it is placed to the bottom of the query.

Listing 6 presents the optimization executed by Sesame for the SeRQL query language. Sesame applies some general optimization rules too. In fact, it rewrites expressions in SeRQL WHERE clause as our optimization framework for FILTER expressions. Further, Sesame reorders SeRQL

Listing 5: Optimized Query

```
PREFIX  opus:  <http://lsdis.cs.uga.edu/propono#>
PREFIX  rdfs:  <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?author ?volume ?pages ?number
WHERE {
        ?article opus:journal_name "VLDB J." .
        ?article opus:year 2004 .
        ?article opus:number ?number .
        ?article opus:pages ?pages .
        ?article opus:volume ?volume .
        ?article rdfs:label ?label .
        ?article opus:publication_authored_by ?author .
}
```

triple patterns defined in FROM clause according to the number of variables. Patterns with more variables are considered to be less specific. Thus, they are executed later in query. This is a more naive approach compared to ours, but it is based on the same idea of reducing the intermediate result set sizes.

By looking at the query (Listing 6), we may notice that the triple pattern containing the predicate `opus:journal_name` should be executed first, since the number of articles published by the VLDB journal are less than those published in 2004. Another problem of the approach used by Sesame emerges for the following triple patterns where just the predicate is specified (Listing 6). Since an article has only one title but mostly several authors, the ordering of the patterns is obviously wrong. This behavior arises because of unavailable statistics about the selectivity of triple patterns. Thus, Sesame is missing fundamental information to optimize the query with a more precise ordering of triple patterns. As we will see later in this section, the more precise reordering of our approach makes a significant difference.

4.4 Results

In this section we present our evaluation results for the retrieval task (Listing 4) performed on the sampled SwetoDblp ontology.

Figure 1 shows the absolute values on a logarithmic scale measured for the retrieval task (Listing 4). We evaluate the query for ARQ, KAON2 and Sesame, where Sesame is evaluated for both SPARQL and SeRQL query languages. The values are measured for each sample. The approximatively linear behavior is expected because of the linear distribution for the resources which matches the retrieval task. Because of main memory limitations of our test server, the measurements for KAON2 stop at 70%. For KAON2, the sample with 551 MB requires over 2.5 GB, which is the maximum amount of memory we can allocate for the Java virtual machine. Please note that the difference between OptARQ and Sesame SeRQL is considerable. In fact, at 100% OptARQ is 31.22 times faster than Sesame SeRQL. The main difference in optimization between OptARQ and Sesame SeRQL consists in the selected approach for triple pattern reordering. While Sesame

Listing 6: Sesame SeRQL Optimized Query

```
SELECT
        Title , Author , Volume , Pages , Number
FROM
        {Article} opus:year {2004};
                  opus:journal_name {"VLDB J."};
                  opus:publication_authored_by {Author};
                  rdfs:label {Title};
                  opus:volume {Volume};
                  opus:pages {Pages};
                  opus:number {Number};
USING NAMESPACE
        opus = <http://lsdis.cs.uga.edu/propono#>,
        rdfs = <http://www.w3.org/2000/01/rdf-schema#>
```

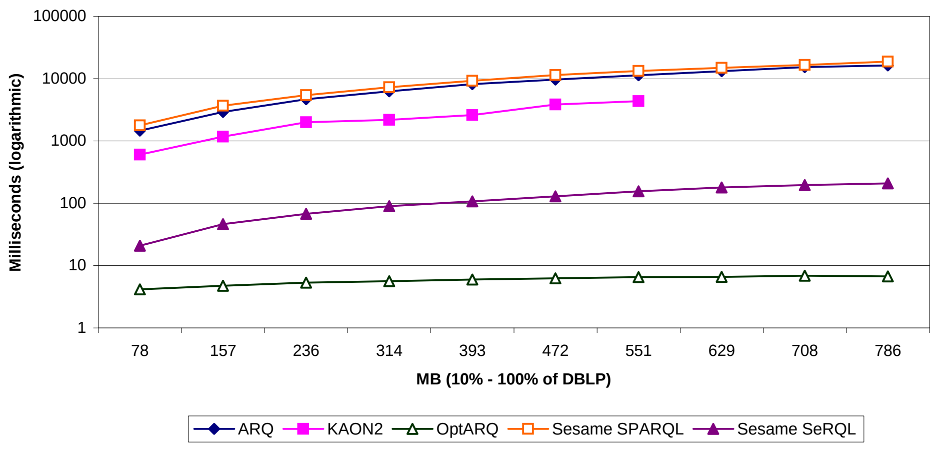

Figure 1: Absolute Values (logarithmic scale)

SeRQL is using a more simple and general approach (i.e., no statistics are required), the model based on statistical information about the underlying ontology used in our optimization framework yields a more accurate reordering. Table 3 shows the measured values for each engine. All values are listed in milliseconds and approximated to the second decimal place.

In order to quantify the performance improvement of OptARQ compared to the other engines, we create a chart which shows how many times OptARQ is faster compared to the other engines. We call the chart 'OptARQ normalized'. Figure 2 shows the factor for ARQ, KAON2 and Sesame on a logarithmic scale.

Finally, we evaluate the performance for each engine when the optimized query is used as input query. Thus, instead to execute the query defined in Listing 4 we execute the optimized query defined in Listing 5. Figure 3 shows the evaluation for the optimized query. Sesame SeRQL outperforms the other engines. The SPARQL implementation in Sesame resulted to be less efficient. Further, the figure shows OptARQ less efficient than ARQ which is straightforward because of the overhead due to the optimizer (which is executed although the query is already optimized). Table

Sample (MB)	OptARQ	Sesame SeRQL	KAON2	ARQ	Sesame SPARQL
78	4.16	20.87	606.20	1’459.59	1’771.45
157	4.76	46.39	1’169.09	2’920.37	3’669.46
236	5.34	67.77	1’997.22	4’668.49	5’444.43
314	5.61	89.90	2’182.24	6’252.73	7’251.89
393	6.00	106.87	2’611.55	8’122.98	9’239.51
472	6.26	129.58	3’841.20	9’641.68	11’454.73
551	6.55	155.05	4’355.03	11’291.75	13’241.42
629	6.57	179.12		13’110.01	14’915.84
708	6.89	196.30		15’247.22	16’575.31
786	6.71	209.46		16’243.94	18’697.85

Table 3: Absolute Values

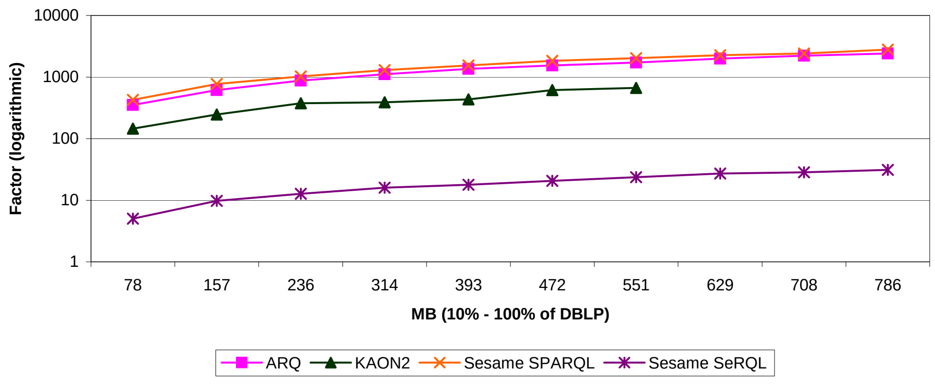

Figure 2: OptARQ Normalized Values

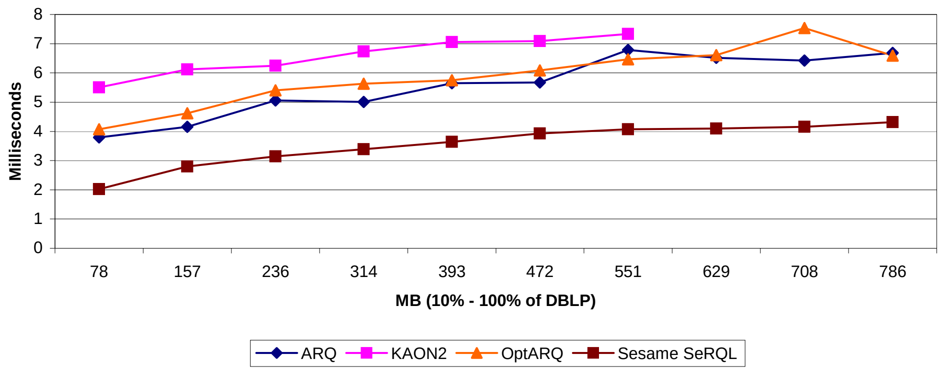

Figure 3: Optimized Retrieval Task

4 lists the absolute values measured for this evaluation including those for Sesame SPARQL[20]. All

[20]We think, Sesame SPARQL performs not better even if the optimized query is used because it does not drop potential results as soon as they don’t satisfy the query pattern. I conclude this because of a separate evaluation performed for Sesame SPARQL where the optimized query for retrieval task A listed in 5 was executed triple pattern by triple pattern. The evaluation showed that Sesame SPARQL performed similar to KAON2 and better than ARQ and OptARQ for the query containing only the first triple pattern. For the query containing the first two patterns Sesame SPARQL performed already 3.5 times inferior to OptARQ at 10%. With three triple patterns the factor is 12.6 at 10%.

values are listed in milliseconds and approximated to the second decimal place.

Sample (MB)	OptARQ	ARQ	Sesame SeRQL	KAON2	Sesame SPARQL
78	4.07	3.80	2.02	5.50	350.41
157	4.62	4.15	2.79	6.12	1’073.65
236	5.40	5.06	3.15	6.25	1’961.03
314	5.63	5.01	3.39	6.73	2’431.63
393	5.75	5.65	3.64	7.05	3’211.96
472	6.09	5.67	3.93	7.09	4’369.13
551	6.46	6.79	4.07	7.34	4’812.85
629	6.61	6.52	4.10		6’095.34
708	7.53	6.42	4.16		6’956.67
786	6.59	6.68	4.31		8’303.01

Table 4: Optimized Retrieval Task

To the best of my understanding, I believe that the engines evaluation with the optimized query demonstrates the correctness of our approach and the optimization techniques discussed in this paper. Since the engines behaves very similar when the optimized query is used, I believe that the performance improvement which is achieved by the optimization techniques is realistic.

Figure 4 shows the improvements between the original query for the retrieval task (Listing 4) and the corresponding optimized query (Listing 5) for each engine on a logarithmic scale. Please note the performance of Sesame SPARQL. Although the optimized query is also performing better for Sesame SPARQL, the performance improvement becomes smaller when the sample size grows.

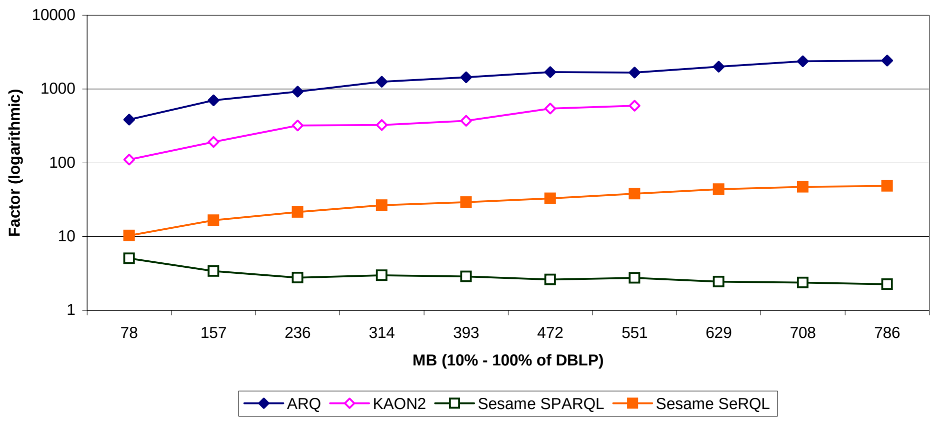

Figure 4: Improvements

5 Limitations and Future Work

The proposed SPARQL optimization framework implements a basic statistical model used for selectivity estimation. More research should be invested to improve the accuracy of estimations. For example, SPARQL variables constrained by an inequality operator (e.g., $>$, $<$) are not considered yet while estimating the selectivity of patterns. Moreover, our framework calculates the object selectivity according to a specific predicate. This limitation can be avoided by generalizing the function for the selectivity estimation of triple pattern objects. In fact, when a predicate is unbound, we may compute the selectivity of an object by considering the histograms describing the object domain values for each predicate. Furthermore, the subject selectivity estimation formula used in our framework may be determined more accurately too. In fact, the subject selectivity is constant in our framework. A more precise statistic where the number of predicates are modeled for each resource class inside the ontology may lead to a more accurate subject selectivity estimation. For example, resources of a class `Person` may have more predicates compared to resources of a class `Address` in an ontology. A statistical representation of the average number of predicates for resources according to the resource class may lead to more accurate and natural subject selectivity estimation. Last but not least, our framework does not consider the behavior of triple pattern selectivity for joined variables. In fact, assigning the selectivity 1.0 to variables which are previously bound is a too rough approximation.

During the last decades, different statistical models have been proposed to characterize attribute value distributions. Our model uses an equal-width histogram to represent the object value domains. Other histogram based approaches have been proposed and are summarized in [2]. They all aim the cost estimation of query execution plans. It has been shown that some methods are less erroneous on selectivity estimation compared to others. In fact, the equal-width histogram based selectivity estimation used in our optimization framework is a relatively simple approach which may lead to significantly higher estimation errors (i.e., large classes result into inaccurate estimation). Moreover, other selectivity estimation models have been proposed too, e.g., probabilistic selectivity estimation models [8].

6 Related Work

Perz *et al.* [5] conduct an extensive analysis of the semantics and complexity of SPARQL, focusing, as argued by the authors, on the two most complicated operators in SPARQL, `UNION` and `OPTIONAL`. This work may be a starting point for discussions about iSPARQL optimizations especially for future optimization rules, since we do currently not consider query optimization for queries including SPARQL `OPTIONAL` or `UNION` keywords.

Sirin [6] presents optimization techniques for OWL-DL ontologies focusing on knowledge bases containing large number of individuals. Aduna Software[21], developer and maintainer of Sesame open source RDF framework[22], introduced some general query optimization techniques based on query rewriting rules for Sesame RDF Query Language (SeRQL).

KAON2[23], an infrastructure for managing OWL-DL ontologies, introduces algorithms which allow optimization of DL reasoning by applying deductive database techniques. According to [7] such algorithms yield to significant performance improvement compared to other available DL reasoners.

[21]http://www.aduna-software.com/
[22]http://www.openrdf.org/
[23]http://kaon2.semanticweb.org/

7 Conclusions

To the best of our knowledge, the proposed SPARQL optimization framework is a first approach for triple pattern selectivity estimation based on statistical information about the resources contained in the underlying ontology. As the evaluation shows, the approach seems to be reasonable and we believe this is the way to go. Obviously, more research work is required to get even more accurate estimations. It is remarkable that a few optimization rules which all aims the common goal to reduce the intermediate result set size of triple patterns highly affect query execution performance.

The optimization work discussed in this paper, focuses on static query reordering in order to get an execution plan which is optimal according to the selectivity of triple patterns. Static optimization techniques may be combined with dynamic techniques to achieve optimization also when static techniques do not lead to any effective optimization (e.g., when the query is already optimized according to the selectivity of triple patterns).

Bibliography

[1] P. Griffiths Selinger and M. M. Astrahan and D. D. Chamberlin and R. A. Lorie and T. G. Price. Access Path Selection in a Relational Database Management System. SIGMOD ’79: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, 1979.

[2] B. Oommen and L. Rueda. The Efficiency of Modern-day Histogram-like Techniques for Query Optimization. 2001.

[3] Gregory Piatetsky-Shapiro and Charles ConnellGregory Piatetsky-Shapiro and Charles Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. SIGMOD ’84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. 1984. ACM Press.

[4] Donald D. Chamberlin and Daniela Florescu and Jonathan Robie and Jérôme Siméon and Mugur Stefanescu. XQuery: A Query Language for XML. 2001.

[5] Jorge Perez and Marcelo Arenas and Claudio Gutierrez. Semantics and Complexity of SPARQL. 2006.

[6] Evren Sirin and Bernardo Cuenca Grau and Bijan Parsia. From Wine to Water: Optimizing Description Logic Reasoning for Nominals. March 2006.

[7] B. Motik and U. Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. November, 2006.

[8] Lise Getoor and Benjamin Taskar and Daphne Koller. Selectivity Estimation using Probabilistic Models. SIGMOD ’01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data. 2001. ACM Press.