RDFMatView: Indexing RDF Data using Materialized SPARQL queries

Roger Castillo, Christian Rothe, and Ulf Leser

Humboldt University of Berlin
{castillo,rothe,leser}@informatik.hu-berlin.de
http://www.hu-berlin.de/

Abstract. The Semantic Web aims to create a universal medium for the exchange of semantically tagged data. The idea of representing and querying this information by means of directed labelled graphs, i.e., RDF and SPARQL, has been widely accepted by the scientific community. However, even when most current implementations of RDF/SPARQL are based on ad-hoc storage systems, processing complex queries on large data sets incurs a high number of joins, which may slow down performance. In this article we propose materialized SPARQL queries as indexes on RDF data sets to reduce the number of necessary joins and thus query processing time. We provide a formal definition of materialized SPARQL queries, a cost model to evaluate their impact on query performance, a storage scheme for the materialization, and an algorithm to find the optimal set of indexes given a query. We also present and evaluate different approaches to integrate materialized queries into an existing SPARQL query engine. An evaluation shows that our approach can drastically decrease the query processing time compared to a direct evaluation.

Key words: SPARQL, Indexing, RDF, Materialized Queries, Semantic Web, Query Processing

1 Introduction

The *Semantic Web* as an evolution of the *World Wide Web* aims to create a universal medium for the exchange of data where data can be shared and processed by automated tools as well as by people. The basis for this proposal is a logical data model called Resource Description Framework (RDF) [1]. An RDF data set is a collection of statements called *triples*, of the form *(s,p,o)* where s is a subject, p is a predicate and o is an object. Each triple states the relation between subject and object. A set of triples can be represented as a directed graph where subjects and objects represent nodes and predicates represent edges connecting these nodes. The SPARQL query language is the official standard for searching over RDF repositories [2].

The increasing amount of RDF data has motivated the development of approaches for efficient RDF data management. Therein, SPARQL implementations have been built either over relational database technology or using an ad-hoc storage system (e.g. Jena [3], 3Store [4, 5], Sesame [6]). Furthermore, very

large scale systems have been proposed using the common paradigm of a triple table normalized using two or more tables (4Store [7], YARS [8]). Consequently, in these systems, joins are still used extensively to answer queries. Optimizing these joins is one of the critical issues to obtain scalable SPARQL systems.

In relational databases, query processing using materialized views is a well established method to achieve scalability [9]. Here, we propose the use of materialized SPARQL queries to speed-up queries. We target large data sets and SPARQL queries consisting of many basic graph patterns producing a set of results. Examples of huge data sets are for instance, UniProt containing more than 600 million triples [10] or the W3C SWEO Linking Open Data Community with more than 4 billion triples [11]. With such datasets, executing a query with many graph patterns becomes a problem.

Listing 1. Example SPARQL query to gather information about Hexokinase enzyme [12]

```
SELECT * WHERE {
   ?s1 ?p1 ?o1 .
   ?o1 ?p2 "hexokinase" .
   ?s1 rdf:type ?type1 .
   ?s1 rdfs:comment ?comment1 .
   ?s1 rdfs:label ?label1 .
   ?s1 rdfs:comment ?comment2 .
   ?s1 rdfs:label ?label2 .
}
```

Consider the query in Listing1. Executing this query on a conventional SPARQL processor over a large triple table results in the computation of six self-joins. However, one can safely assume that the types, labels, and comments of an object are used together very often. Therefore, similar to [3], which create tables to group properties that tend to be defined together we suggest to cluster frequently used *triple patterns* by materializing and storing the results inside the system. If this information were available, the query could be computed with only three joins, as the materialized query would help to retrieve the information for $s1$.

Our indexing method aims to fully exploit the RDF graph-structure. We do not index single attributes or triples, but fractions of queries that occur frequently in an expected workload. Therefore, our approach is a *native RDF/SPARQL indexing* method whose concepts are viable for all possible implementations of RDF stores. Our method can be seen as an orthogonal indexing solution, which may be used in conjunction with other indexing methods.

Such an approach requires to solve several problems. First, selected queries must be materialized and the results stored such that efficient retrieval is possible. Second, a given query at runtime must be analyzed to identify the materialized query or the combination of materialized queries that offers the highest speed-up for this query. This requires a query rewriting algorithm and a cost model. Third, the query processing itself must be modified to be able to retrieve materialized results and to combine them with those parts of the original query that are not covered by the indexes.

Here, we present solutions to these problems. We first discuss related work in Section 2. Section 3 presents the fundamental principles of *RDFMatViews*. We describe different ways to introduce materialized queries into an existing SPARQL processor in 4. Section 5 gives an evaluation of our method. We conclude in Section 6.

2 Related work

In the following, we discuss those works that are most related to our main contribution, i.e., using indexes to speed up SPARQL queries.

Some approaches have proved to be very efficient to query SPARQL queries based either on relational database technology or following a native data scheme. For instance, in [13] Abadi et al. propose a vertical partition approach for Semantic Web data management. An enhancement of this approach is proposed by Weiss et al. in [14]. Therein, RDF data is indexed in six possible ways, i.e., an index for each possible ordering of the three RDF elements. Each instance of an RDF element is associated with two vectors; each such vector gathers elements of one of the other types, along with lists of the third-type resources attached to each vector element. This scheme is capable of speeding up single joins tremendously, but storage requirements are very high, which becomes a serious issue when using huge data sets.

Neumann and Weikum developed *RDF-3X*, a SPARQL engine pursuing a RISC-style architecture – a streamlined architecture – with specific-designed data structures and operations [15]. The authors overcome the "giant-triples-table" [13] bottleneck by creating a set of indexes and a fast way for processing merge joins. Similar to [14], RDF-3X maintains six possible permutations of subject, predicate and object in six separate indexes. The authors also present a compression algorithm to decrease the space consumption.

All these approaches have in common that they focus on indexing the relational representation of the RDF data. When faced with queries consisting of multiple basic graph patterns, they still have to compute multiple joins (although every single join is faster). In contrast, our work specifically targets the speed-up of complex queries consisting of many basic graph patterns by indexing complete query patterns which occur in other queries.

There is some other work along this line. In [16] the authors present *GRIN*, a lightweight indexing mechanism for RDF data. The idea is to draw circles around selected *center* vertices in the graph where the circle would comprise those vertices in the graph that are within a given distance of the "center" vertex. Basically, GRIN is a binary tree where the set of leaf nodes form a partition of the set of triples in the RDF graph. An interior node represents the set of all vertices in the RDF graph that are within a specific distance. To evaluate a query, GRIN derives a set of inequality constraints from the query. These constraints are evaluated against the nodes of the GRIN index.

A similar indexing approach is presented in [17]. This work proposes a set of indexes of precomputed joins created from all possible join combinations be-

tween triple patterns. As [16], this approach creates a general purpose set of indexes based on joined triple patterns, but the number of indexes to manage is impractical when the number of joined triples is >= 3.

The two systems just described index larger portions of the RDF data set and not just single triples. However, they propose to apply their techniques to all RDF triples, while we only build user-chosen indexes. Our work fundamentally is based on the assumption that some patterns are combined more frequently than others, and that only indexing those combinations promises to provide large speed-ups at manageable space and maintenance cost.

The differences between our ideas and that of other RDF indexing schemes can be described by drawing a parallel to B*-indexes in relational databases [18]. Nobody would suggest to speed up queries by indexing every attribute; instead, systems assume that developers have a rough idea about the types of queries that need to be answered and therefore index only the relevant attributes. Furthermore, optimal speed-up can only be achieved when also combinations of attributes can be indexed, and not only single attributes. In this sense, the former approaches index every single attribute, the latter indexes every possible combination of attributes, and we suggest to index only selected combinations of attributes.

Note that we do not claim that our current implementation of RDFMatView offers a particular fast SPARQL-processor, compared to systems such as [13–15]. Instead, we present a new technique to speed up query execution with SPARQL that is applicable to any SPARQL query processor. We showcase its potential and compare different ways to integrate it into query processing using one particular system (namely ARQ [19]), which was been chosen because of its widespread use. An extended version of this paper can be found in [20].

3 The RDFMatView Approach

3.1 Indexes and Covers

A SPARQL query Q is represented by a simple graph pattern P and is denoted by $P(Q)$. A mapping is a function that maps the symbols of one pattern into the symbols of another pattern. Our notion of mappings is based on the SPARQL-Standard [2] and its definition of *pattern solutions*. However, while in the SPARQL standard such solutions are only searched in the data graph, we also permit that variables are mapped to other variables. This generalization allows us to search occurrences of patterns in other patterns, in particular, occurrences of indexes in a query. We say that a pattern P_1 occurs in a pattern P_2 if there is a mapping function S such that $S(P_1) \subseteq P_2$. Extending occurrences and mappings also to RDF triples, we define an index over an RDF data graph as follows.

Definition 1 (Index). *An* index *I over a data graph G is a pair $I = (P, O)$, where P represents a pattern and O represents the set of all occurrences of P in G.*

Indexes are precomputed queries suitable to speed up other queries when the index pattern is "contained" in the query pattern. An index I is eligible for a query Q when the patterns set of I occurs in the pattern set of Q. However, it would be more advantageous when query processing uses more than one index. For those cases, we require that indexes "overlap". Overlapping indexes are good candidates for reducing query processing because the query engine can combine occurrences of these indexes and generate solutions without matching against the RDF dataset.

We define two ways in which indexes overlap. Two indexes overlap intensionally iff there *could* exist a triple pattern in which their materialization would overlap. Two indexes overlap extensionally if their materializations overlap on a concrete data graph. Thus, intensional overlap relies only on the index patterns and is independent of a concrete data graph, while extensional overlap needs to consider the actual data graph.

Definition 2 (Overlapping Indexes). *Let $I_1 = (P_1, O_1)$ and $I_2 = (P_2, O_2)$ be two indexes over a data graph G.*

- *I_1 and I_2 intensionally overlap iff there exists mapping functions S_1, S_2 such that*
$$S_1(P_1) \cap S_2(P_2) \neq \emptyset$$
- *I_1 and I_2 extensionally overlap in G iff*
$$\exists o_1 \in O_1, o_2 \in O_2 : o_1 = o_2$$

where o_i is a concrete occurrence in O_i.

Here, we only consider intensional overlapping since it is independent from the data graph and can be efficiently implemented. Using intensionally overlapping indexes, we define a cover. Think of set E which contains all the eligible indexes as graph nodes, connected by an edge when they overlap. For a given query Q, we call every subset of E a *cover of Q*, for which the induced subgraph has a single component. Furthermore, we are only interested in *maximal covers*, i.e., those covers which cannot be extended further by adding new indexes.

3.2 Example

Finding covers requires to analyze the set of indexes and the given query. The idea is to find mappings between index and query patterns. This process is performed using a query containment algorithm [21] adapted for SPARQL queries. Details of the algorithm can be found in [20]. Essentially, we find all mappings between any index pattern and the query pattern by enumerating all possible cases. If a mapping exists, the index is eligible for that query and we store the mapping. Note that, for a given index, there are potentially many different ways to be eligible, i.e., different mappings between index and query patterns. Consider a SPARQL query and two indexes described in Table 1.

Table 1. SPARQL query returning universities and their departments, Index1 computes places and their names and Index2 computes universities with their departments.

```
Query: SELECT * WHERE {
       ?university  rdf:type ub:University;
         ub:name ?university_name.
       ?ub_department rdf:type ub:Department;
         ub:name ?ub_name_department;
       ub:subOrganizationOf ?university . }
Index1: SELECT * WHERE {
         ?place  rdf:type ?place_type;
             ub:name ?place_name. }
Index2: SELECT * WHERE {
         ?ub_department rdf:type ub:Department;
             ub:name ?ub_name_department;
             ub:subOrganizationOf ?university;
         ?university  rdf:type ub:University.}
```

Index1 and Index2 are eligible for the query, using the mappings in Table 2. Note that Index1 has two mappings. Each mapping represents an occurrence of the patterns of Index1 in the query pattern. Index occurrences generated from the previous mapping functions overlap in the triple pattern *?university rdf:type ub:University*, using the first mapping function of Index1. Hence, partial results can be joined to completely cover the query.

Table 2. Mappings of Index1 and Index2

Index1: Mapping 1			**Mapping 2**		
?place	⇒	?university	?place	⇒	?ub_department
?place_type	⇒	ub:University	?place_type	⇒	ub:Department
?place_name	⇒	?university_name	?place_name	⇒	?ub_name_department
Index2: Mapping 1					
?ub_department	⇒	?ub_department			
?ub_name_department	⇒	?ub_name_department			
?university	⇒	?university			

Assume an RDF data stored in a RDBMS within a *triple* table (without indexes), for instance, *Triple(subj, prop, obj)* [3, 15]. Hence, query in Table 1 could be answered by the SQL query in Listing 2 requiring four self joins. However, using pre-computed tables, Index1 and Index2, requires only one join as shown in Listing 3.

Listing 2. SQL representation of SPARQL query in Table 1

```
SELECT t1.subj AS a0, t2.obj AS a1, t3.subj AS a2,t4.obj AS a3
FROM Triple AS t1, Triple AS t2, Triple AS t3,
     Triple AS t4, Triple AS t5
WHERE t1.prop= 'type' AND t1.obj= 'University' AND
      t2.prop= 'name' AND t3.prop= 'type' AND
      t3.obj= 'Department' AND t4.prop= 'name' AND
      t5.prop= 'subOrganizationOf' AND
      t1.subj = t2.subj AND t3.subj = t4.subj AND
      t3.subj = t5.subj AND t1.subj = t5.obj;
```

Listing 3. SQL representation of SPARQL query in Table 1 using RDFMatView indexes

```
SELECT index2.university, index1.place_name AS university_name,
       index2.ub_department, index2.ub_name_department
FROM index1, index2
WHERE index1.place = index2.university;
```

This example illustrates advantages to use materialized queries as indexes to process SPARQL queries. Note that this case does not require to query against the data graph, because all query patterns are covered and the partial results are materialized. Other cases would require to extend the results of the covered patterns with the results of query patterns which are not covered using indexes.

3.3 Cost Model

Previous sections define which indexes and which sets of indexes are eligible for a given query. In the following, we define a model to estimate which cover brings more savings in time to query execution. Our model is based on the definition of *selectivity* of an index. Selectivity defines the relation of the number of index occurrences in a given graph to the possible total number of index occurrences in the graph. To this end, we need the *size and frequency of the index pattern* (number of triples in the index pattern and number of tuples for the index pattern in the data graph) and the *size of the data graph* (total number of triples) represented by $|I|$, $\#(I)$ and $|G|$ respectively. Hence, we define selectivity as follows.

Definition 3 (Selectivity of an index). *Let I be an index over a data graph G. The selectivity $s(I)$ of an index I is defined as:*

$$s(I) = \frac{\#(I)}{|G|^{|I|}}.$$

We can estimate selectivity of two indexes based on the overlapping of their index patterns, i.e., they can completely, partially or not overlap at all. This leads to estimate the selectivity of a cover.

Definition 4 (Selectivity of a cover). *Let C be a cover for a query Q consisting of indexes $I_1, I_2, \ldots I_n$. The selectivity $s(C)$ of the cover C is defined as:*

$$sel(C) = sel(I_1 \cup I_2 \cup \ldots \cup I_n) \leq \frac{min|O_1|, ..., |O_n|}{|G|^{\max\{|P_1|,...,|P_n|\}}}$$

Having the selectivity of all maximal covers, the query optimizer determines which cover is the best for query processing.

4 Implementation

We describe the implementation of our approach into a SPARQL query processor by using the ARQ system [19]. However, we want to stress that the general process would be the same for any other SPARQL query processors. We differentiate two main phases in our approach. At offline-time, indexes are created and materialized. At query-time, queries are answered using indexes (or covers). We describe our implementation regarding these phases.

4.1 RDFMatView Index processing

Each index is preprocessed as a table in the underlying database. We create its schema by materializing all variables of the index and not only those mentioned in the SELECT clause. This strategy enables the materialized data to be eligible also for those queries requesting variables that were not selected in the original index. Occurrences of the index in the dataset are stored as values for these fields. Each attribute of a tuple represents a binding for the respective variable. To avoid large requirements of storage space, we use an *RDF Data Dictionary*, which maps each resource to a unique identifier. Thus, instead of storing large literal values, we store only numeric identifiers in the index structure. Table 3 summarizes the space required to store 10 indexes for each RDF dataset. During the processing of an index we also calculate and store index properties, for instance size and frequency, which are later used to evaluate the query execution plans. These tasks are executed only once per index and can be used to process any SPARQL query.

Table 3. Storage required for a set of 10 RDFMatView indexes over 4 different databases (BSBM) [22] (K = 1000, M = 1 Million triples).

	$250K$	$500K$	$1M$	$10M$
index storage	12Mb	18Mb	34Mb	363Mb
total storage	379Mb	616Mb	1.2Gb	12Gb
storage ratio	0.03	0.02	0.02	0.03

4.2 Executing a query using RDFMatView indexes

Query processing using *RDFMatViews indexes* usually combines results of multiple indexes. However, it is not always possible to cover all query patterns. The set of uncovered patterns is referred to as *residual part of a query*. This breaks down into the following steps: i) Analysis of the query to find all maximal covers ii) Selection of the most suitable cover to answer the query given our cost model iii) Rewriting of the query using the chosen cover iv) Extension of the results of the cover to results of the query. Steps one and two were already discussed in Sections 3.1 and 3.3 respectively. Here, we concentrate on the two last steps, the query rewriting. We developed three strategies to fulfill this task:

- Our first strategy uses ARQ to process the residual part of the query. RDFMatView extends the results of the chosen cover by joining the partial solutions with the solutions of the residual patterns.
- The second strategy is based on a SPARQL-to-SQL algorithm to translate SPARQL queries into SQL queries. The idea is to directly access the native Jena storage tables and to combine those results with the index tables to generate the final solution.
- The last strategy is built from a combination of the previous two strategies, i.e. ARQ and database execution engine.

These strategies are explained in detail in the next sections.

Method 1: MatView-and-ARQ Engine

Rewriting engine built on top of the Jena Framework. Given a query and a cover, it computes the set of uncovered residual patterns of the query and uses ARQ to execute this (sub-)query. Furthermore, it computes the result of the cover by joining the respective index tables according to the variable mappings between the query and the indexes forming the cover. Results are also joined with the data dictionary to obtain RDF values, and joined to the result of the ARQ query to produce the complete answer for the original query. This engine encapsulates the logic for the execution of the cover and provides total independence from the underlying database.

Method 2: MatView-to-SQL Engine

Rewriting engine which, unlike our first method, translates the residual part of the query into a SQL query using an algorithm proposed by Chebotko in [23]. The SQL query is executed by the RDBMS which evaluates the query using the Jena tables. The result set is processed using our dictionary and combined with the results of the cover. The complete query processing is performed by the database execution engine using a stored procedure.

Method 3: Hybrid Engine

A mixture of MatView-and-ARQ and MatView-to-SQL. As in Method 1, after rewriting the query, this engine transfers the residual patterns to the query execution engine of ARQ. The second part of the process combines the results of the residual patterns with the resulting set of the covered part of the query patterns. Contrary to Method 1, this engine is database-dependent since this task is performed inside the database execution engine, as in Method 2.

5 Evaluation

We evaluate our approach using two well known SPARQL benchmarks: the Berlin SPARQL Benchmark (BSBM) [22] and the SPARQL Performance Benchmark (SP^2B) [24]. We use the ARQ/Jena RDF Storage System (version 2.5.7) on Postgres 8.2 as framework in which we integrated our solution. We generated eight RDF datasets with sizes ranging from 250K to 10M triples and tested the impact of the indexes on six different queries (three queries for each benchmark). For each query, we manually defined a set of indexes, leading to covers composed of one to three indexes. Our intention here is not to find the best set of indexes given a workload (generally called index selection, see, e.g., [25–27]); instead, we study to which degree indexes that use different processing schemes speed up the execution of queries.

5.1 Dataset and queries

For each benchmark, we create four datasets containing 250K, 500K, 1M and 10M triples, respectively. As these datasets have identical value distributions but different sizes, our evaluation concentrates on the scalability of our methods in different domains. Based on the number of triple patterns we chose three queries for each benchmark. We transformed the query patterns into simple graph patterns and removed most bindings to variables. Bounded variables incur high selectivity resulting in the retrieval of only a handful of triples. Such queries are well supported by existing index structures and do not require the type of join-optimization that is achieved with our optimization technique. Therefore, performance gains would be only marginal. Our test queries are described in Listings 4 and 5.

Listing 4. Test queries derived from BSBM

```
Query1: Finds products for a given set of generic features.
Query2: Retrieve basic information about products.
Query3: Retrieve in-depth information about products including
offers and reviews.
```

Listing 5. Test queries derived from SP^2B

```
Query4: Extract all information about inproceedings documents.
Query5: Select all pairs of articles of an author that have been
published in the same journal.
Query6: Return for each year, the set of all publications including
the name of the authors.
```

From the queries described in Listings 4 and 5, we derive two sets of indexes containing 10 and 8 indexes respectively. Each index covers two to six patterns from at least one query. However, none of them completely covers a query. We focus to evaluate covers containing either a combination of indexes and possible a residual part of the query since most real-life SPARQL queries would comply with this case.

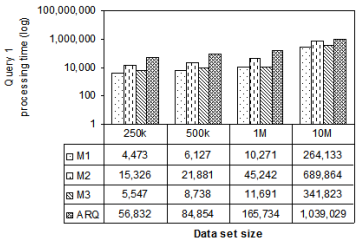
5.2 Results

For each benchmark we evaluated three queries over four data sets using our three RDFMatView methods and plain ARQ (without indexes), which amounts 36 different configurations. We refer to the approaches to query execution as M1 for MatView-and-ARQ, M2 for MatView-to-SQL, M3 for the hybrid approach, and ARQ for plain ARQ. The experiments use the optimal cover and evaluate the real and estimated costs of different covers for the same query. All queries were executed 5 times and average execution times are reported[1]. Furthermore, we evaluated all different covers generated for Query1 and Query2 (BSBM) to show the performance of our cost model in the selection of the query execution plan. Figure 1 illustrates the average processing time for each query. Clearly, processing time significantly improves in both domains when using MatView-and-ARQ (M1) and Hybrid (M3). However, processing time does not significantly improve when using MatView-to-SQL (M2) (see Fig. 1). The reason for this is the Jena native storage schema. Since the values are encoded following the Jena layout, our process needs to parse the stored values and extract the required information, which increases the processing time.

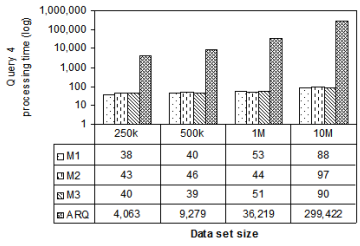
Figure 2(a) and Figure 3(a) show the evaluation of real and estimated cost for different covers for Query1 and Query2 (BSBM). Additionally, Figure 2(b) and Figure 3(b) show the relation between the estimated costs of a cover, its indexes and number of covered and uncovered patterns from the given query. Note that our system selects as optimal Cover 6 in Figure 2(a) (for Query1) and Cover 3 in Figure 3(a) (for Query2).

Figure 2(a) and Figure 3(a) show the costs estimated by our model together with the real processing time. In all cases our model manages to prevent the selection of exceptionally bad plans, and all plans improve the total execution times when compared to those without using indexes. However, the figures also show that our model can be improved further, as total real and estimated costs do not correlate well. Especially, our model does not yet reflect the fact that, in

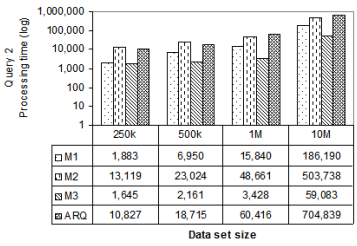
[1] Except for Query5 without indexes over a 10 Million triples dataset, which did not finish after 24 hours



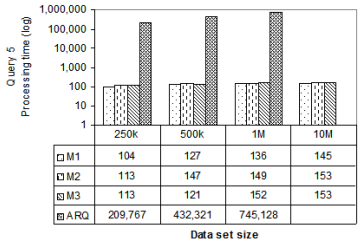
(a) Query1 (BSBM)



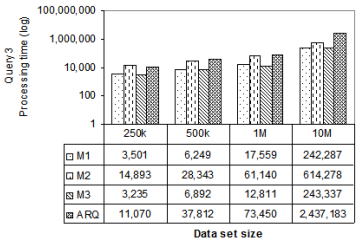
(b) Query4 (SP^{2}B)



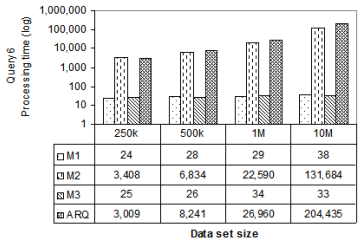
(c) Query2 (BSBM)



(d) Query5 (SP^{2}B)

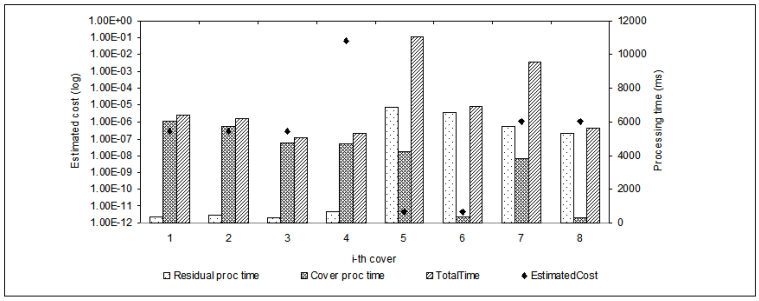


(e) Query3 (BSBM)

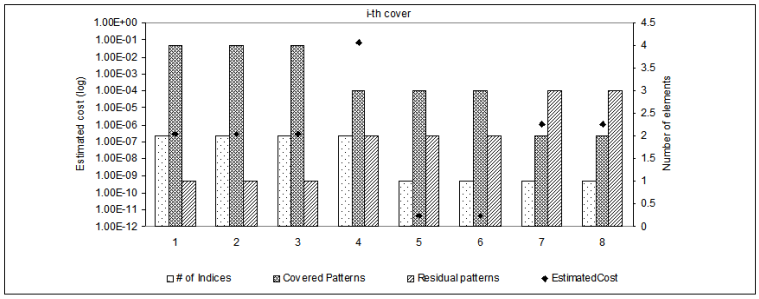


(f) Query6 (SP^{2}B)

Fig. 1. Processing time for test queries using BSBM and SP^{2}B. Each query was processed on four data sets using three rewriting methods. M1: MatView-and-ARQ; M2: MatView-to-SQL; M3: Hybrid; ARQ: plain ARQ (time in milliseconds).



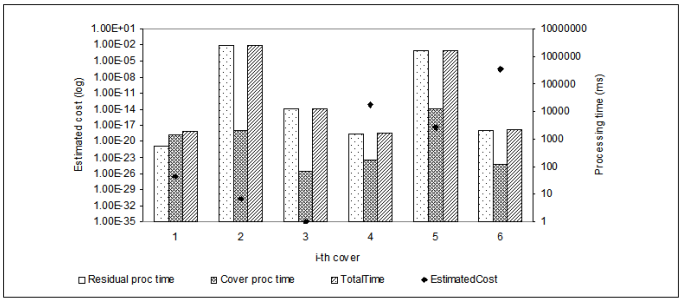
(a) Estimated cost vs. real processing time



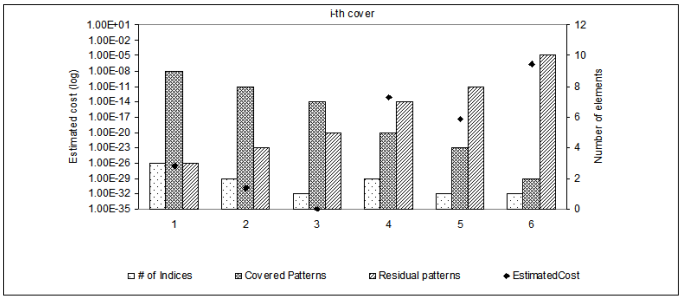
(b) Estimated cost vs elements processing time

Fig. 2. Figure 2(a) shows estimated cost, total real processing time, cover processing time, and residual processing time of Query1. Values are plotted on log-scale. Note that total real processing time virtually equals real processing time for the covers for larger covers. Figure 2(b) shows estimated costs for each cover based on intensional dependency between indexes for the same query. Costs are based on the model introduced in Section 3.3 and are presented in relation to the size for each cover, number of participating indexes and the size of the residual part of the query. This analysis shows the influence of these elements in the selection of an optimal cover.

a setting with two covers both covering the same number of patterns, but with a different number of indexes, it is usually advantageous to chose the cover with less indexes as this requires less joins at runtime. Figure 2(b) illustrates that plans with fewer indexes have a superior performance than plans with the same number of covered patterns, but consisting of more indexes. Thus, the number of necessary joins between indexes is a natural next factor to consider in future work. In Figure 2(b), Covers 5 and 6 have the best estimated costs according to our model. However, the residual part of the query (2 triple patterns) incurs an undesirable overhead, which is not yet properly reflected in our model. An interesting fact can be observed for those covers covering larger patterns using two indexes (see covers 1, 2 and 3). These cases show the reduction of processing time when joining two indexes. At the end, more patterns are covered and the number of patterns to match against the data set decreases. Though their cost



(a) Estimated cost vs. real processing time



(b) Estimated cost vs elements processing time

Fig. 3. Figure 3(a) shows estimated versus real cost for Query2. Estimated costs correlate with cover real processing time however, residual processing time consumes most of the real processing time. Figure 3(b) shows for Query2 estimated cost versus number of covered patterns and number of indexes; for explanation, see Figure 2(b).

estimation is not the best, their processing times are significantly better than those of covers with a better estimated cost. We attribute this behavior to the join (between indexes) and the processing of the residual part of the query which decreases the fewer are the patterns.

As for Query1, Figure 3(a) shows that the estimated costs and the real processing time for Query2 approximately correlate. Additionally, the graphic shows that the residual part of the query should be considered as an important factor when selecting an optimal cover, since residual processing time nearly spans the complete total processing time. Figure 3(b) supports this conclusion showing the details for the generated covers, i.e., covering a larger number of query patterns using as few indexes as possible decreases the real processing time.

6 Conclusions and future work

In this article we proposed a logical framework and a prototype implementation for answering SPARQL queries using materialized queries as indexes. At runtime, queries are analyzed to see whether their execution can be sped-up by using one

or more of those precomputed partial results. The subsequent query rewriting and integration of precomputed results into the overall result generation was implemented following three different approaches on a standard SPARQL query processor. Initial experiments with different queries, different indexes, and different data sets showed that the performance gains in query processing can be considerable.

However, a closer look reveals that our cost model needs to be improved in several aspects. In particular, it needs to model the influence of the number of used indexes, the size of the covers, and the number of residual query patterns. A more accurate estimation of the impact of these elements and its inclusion in the cost model is important to select an better cover.

Up to now, we assume a predefined set of indexes suitable for a given workload of SPARQL queries. A natural extension to this assumption is to study ways of finding the optimal set of indexes under some resources constraints, given a workload. We report on some initial results in this direction in [28], but these also need to be improved by using a better cost model.

References

1. Manola, F., Miller, E.: RDF Primer (February 2004) W3C Recommendation.
2. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (April 2008) W3C Recommendation.
3. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: Proc. First International Workshop on Semantic Web and Databases. (2003)
4. Stephen Harris, N.G.: 3store: Efficient Bulk RDF Storage. In: 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03). (2003)
5. Harris, S.: SPARQL Query Processing with Conventional Relational Database Systems. In: International Workshop on Scalable Semantic Web Knowledge Base System. (2005)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: International Semantic Web Conference. (2002) 54–68
7. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). (2009)
8. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: LA-WEB '05: Proceedings of the Third Latin American Web Congress, Washington, DC, USA, IEEE Computer Society (2005) 71
9. Goldstein, J., Larson, P.A.: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In: SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (2001) 331–342
10. Dataset, U.R.: http://dev.isb-sib.ch/projects/uniprot-rdf/
11. Project, W.S.C.: Linking Open Data on the Semantic Web.
12. Bio2RDF. http://bio2rdf.org/ (2009)

13. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management using Vertical Partitioning. In: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 411–422
14. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. Proc. VLDB Endow. **1**(1) (2008) 1008–1019
15. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proc. VLDB Endow. **1**(1) (2008) 647–659
16. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: A Graph Based RDF Index. In: AAAI. (2007) 1465–1470
17. Groppe, S., Groppe, J., Linnemann, V.: Using an Index of Precomputed Joins in order to speed up SPARQL Processing. In Cardoso, J., Cordeiro, J., Filipe, J., eds.: Proceedings 9th International Conference on Enterprise Information Systems (ICEIS 2007 (1), Volume DISI), Funchal, Madeira, Portugal, INSTICC (June 12 - 16 2007) 13–20
18. Connolly, T.M., Begg, C.E., Strachan, A.D.: Database Systems: A Practical Approach to Design, Implementation and Management. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1996)
19. ARQJena: ARQ - A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/ (2010)
20. Castillo, R., Leser, U., Rothe, C.: RDFMatView: Indexing RDF Data for SPARQL Queries. Technical Report 234, Humboldt Universitaet zu Berlin (2010)
21. Halevy, A.Y.: Answering Queries Using Views: A Survey. The VLDB Journal **10**(4) (2001) 270–294
22. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal On Semantic Web and Information Systems - Special Issue on Scalability and Performance of Semantic Web Systems, 2009 (2009)
23. Chebotko, A., Lu, S., Jamil, H.M., Fotouhi, F.: Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical report, Department of Computer Science, Wayne State University (2006)
24. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^{2}Bench: A SPARQL Performance Benchmark. Data Engineering, International Conference on **0** (2009) 222–233
25. Comer, D.: The Difficulty of Optimum Index Selection. ACM Trans. Database Syst. **3**(4) (1978) 440–445
26. Caprara, A., Fischetti, M., Maio, D.: Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design. IEEE Transactions on Knowledge and Data Engineering **7**(6) (1995) 955–967
27. Chaudhuri, S., Narasayya, V.R.: An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In: VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1997) 146–155
28. Castillo, R., Leser, U.: Selecting Materialized Views for RDF Data. In: Semantic Web Information Management Workshop (SWIM 2010). (2010)