Index Support for SPARQL

Ralf Heese, Ulf Leser, Bastian Quilitz, and Christian Rothe

Humboldt-Universität zu Berlin
Department of Computer Science
(rheese|leser|quilitz|rothe)@informatik.hu-berlin.de

Abstract. The Resource Description Framework (RDF) is the fundamental data model underlying the Semantic Web. Recently, SPARQL has been proposed as W3C standard for querying RDF repositories. As RDF is a graph-based data model, the core problem of evaluating SPARQL queries is matching query graph patterns against the data graph, which is computationally very expensive. We address this problem by indexing graph patterns. In the spirit of SQL, we assume that users select the patterns to be indexed. We formally define the problem of covering indexes, i.e., finding those indexes whose graph-patterns are contained in the query pattern, and derive formulas for estimating index selectivity. Finally, we study the problem of finding optimal sets of indexes for a given query. We believe that our framework is the first comprehensive suggestion for indexing RDF for SPARQL queries that goes beyond simple indexing of labels.

1 Introduction

Efficient query processing is a key topic for database management systems (DBMS). The choice of appropriate methods for speeding up queries depends on the query language and on the data model underlying the DBMS. For instance, the relational model together with relational algebra has lead to the development of indexing techniques for individual attributes (such as the B-Tree [1]) and sets of attributes (multi-dimensional index structures [2, 3]), since accessing tuples by the value of attributes is in the core of most relational operators. With the upcoming of the object-oriented model, indexing techniques for paths emerged because following pointers across objects is a fundamental operation for OO queries [4].

Recently, the W3C committee standardized the Resource Description Framework (RDF [5]) as the fundamental data model underlying the Semantic Web. Several RDF repositories have been developed using existing database technologies as storage medium [6–9]. The development of those repositories also fostered research on query languages for RDF, such as RDQL [10] or RQL [11]. These coalesced into the SPARQL Protocol And RDF Query Language (SPARQL), which is the current candidate recommendation of the W3C for querying RDF data [12].

SPARQL is a declarative, graph-based query language. A query essentially is a combination of graph patterns, and the main operation while executing

a SPARQL query is the matching of those basic graph patterns against the underlying RDF database. Most current SPARQL implementations translate a query into a set of relational queries against the underlying relational DBMS, usually transforming triples in the query graph into joins on the underlying triple table. To evaluate these joins, the DBMS uses its conventional relational indexes. We argue that this approach is insufficient for handling SPARQL with relational databases, because it requires as many joins as the query has triple patterns. Instead, we believe that indexing of patterns – rather than indexing attributes – is the better, and more natural, choice.

In this paper, we present basic considerations about pattern-based indexing of RDF databases for SPARQL queries. We adopt the general idea of relational databases that users and administrators know their workload best and therefore should choose appropriate indexes themselves. We define an index as a materialized SPARQL query whose occurrences in the underlying RDF database are computed in advance. We formalize the tasks of the query optimizer for choosing an index or a set of indexes, given a SPARQL query. This leads to the notion of eligible indexes, i.e., indexes whose patterns occur completely in the query pattern. We show how the set of eligible indexes can be computed and develop a formula for estimating the selectivity of eligible indexes for a given query. We also discuss methods for selecting an optimal combination of eligible indexes and for estimating their selectivity. Finally, we sketch how our framework could be integrated into a native SPARQL query processor.

1.1 Related Work

There has been much research on RDF query evaluation over the last years. Current implementations of RDF stores, such as Jena [13] or Redland [14], use relational databases for persistent storage and utilize the indexing capabilities of the underlying database. Yars [15] uses indexes optimized for RDF, but these are also limited to single triple patterns. The restriction of indexing triples results in the loss of structural information. Complex graph pattern queries have to be decomposed into conjunctive queries of triple patterns and the results of these queries are joined. Our work differs from current approaches used in RDF stores, because we support indexes on basic graph patterns.

In [16], the authors propose a query language with well-defined semantics and theoretically study the complexity of query processing, query containment, and simplification of answers. Besides providing a formal foundation our work aims at a practical implementation.

Other related work includes research on indexes for object-oriented and XML databases, and graph indexing. The work for object-oriented databases, e.g., [4], strongly depends on a fixed schema, which cannot be expected when using RDF. For instance, the indexed properties used to navigate from one object to another are fixed, while SPARQL also allows variables in place of properties. Furthermore, OO approaches only index navigation paths between classes and objects.

In the field of XML, numbering schemes have been developed to efficiently index path expressions [17]. These approaches are specialized structures to provide efficient access to trees. The approaches cannot be applied directly to RDF, because RDF data is a directed labeled graph which often contains cycles.

In the context of graph databases, fast algorithms have been developed to solve the subgraph isomorphism problem, for example [18], but their usage is limited to small graphs residing in main memory. Frequent subgraph mining as described in [19] is orthogonal to our approach. Instead of mining for frequent subgraphs we assume that the users and administrators of the RDF repository define the indexes.

Comment 1: In contrast to conjunctive queries, we also allow variables in place of properties.

1.2 Structure of this Paper

In the next section, we briefly review the RDF data model and SPARQL and present the running example. In Section 3, we introduce pattern-based indexes on RDF graphs. Afterwards, we formally define the index selection problem in Section 4. Section 5 discusses the problem of selecting an optimal set of indexes according to a cost function. We describe a solution to the problem and a formula for estimating index selectivity. In Section 6, we sketch how our methods can be included into a query engine for SPARQL, and Section 7 concludes the paper.

2 Preliminaries

Inspired by past proposals for query languages to retrieve data from RDF graphs, the W3C currently standardizes a query language for RDF, namely SPARQL Query Language for RDF [12]. The working draft defines a basic set of language structures to declaratively query RDF graphs. The user can specify the form of the result, the accessed data sources, and restrictions to select information from the data graphs. We assume that the reader is familiar with RDF [5] and SPARQL. Thus, we mention only the most important terms of these specifications in this section.

An RDF graph is defined as a set of triples and a triple in turn consists of RDF terms. An RDF term is either an IRI, a blank nodes, or an RDF literal – we denote with RDF-Tthe set of all RDF terms. We refer the reader to the RDF specification [5] for their formal definitions.

The building blocks of a SPARQL query are triple patterns and filter expressions. A triple pattern is best described as a triple that may contain variables, and a filter expression is a boolean-valued expression. A set of triple patterns is called a basic graph pattern. Complex pattern are constructed by combining basic graph patterns and filter expressions using operations such as `AND`, `UNION`, or `OPTIONAL`.

In this study, we restrict ourselves to conjunctive SPARQL queries, i.e., queries where triple patterns are connected only by logical AND. We will extend

our approach to handle `UNION` and `OPTIONAL` in our future work (see Section 7). Furthermore, we do not consider the form of the result set, e.g., set of variable bindings or RDF graph. Therefore, we use the following definition of a query throughout this paper:

Definition 1 (Query). *A* query *Q is a basic graph pattern.* □

In the remainder of this paper, we refer to the basic graph pattern of a query as *query pattern*.

As our running example, we use the RDF schema shown in Figure 1 to illustrate our ideas. The RDF schema is a part of the RDF scheme used in the Lehigh University benchmark [20].

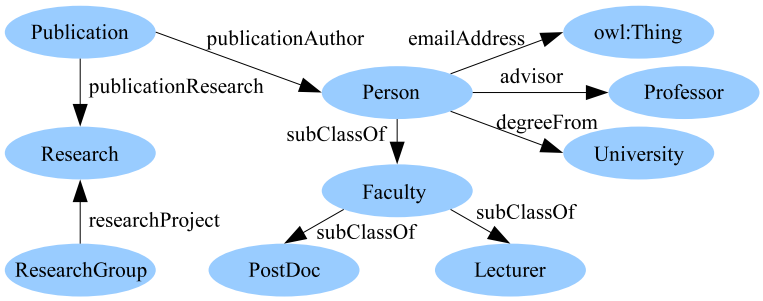

Fig. 1. RDF schema of the running example (namespaces are omitted)

3 Index Support for SPARQL

In database management systems (DBMS) indexes are important access paths to efficiently retrieve database tuples that satisfy a certain constraint. In most cases, it is sufficient to create an index on a single attribute, although current relational DBMSs also provide specialized index structures for non-standard applications such as spatial data or XML documents. Since the data model of RDF bases on graphs and graphs are cumbersome to be represented in relational databases, we believe that specialized index structures are also needed for RDF data.

We assume that users create indexes on a basic graph pattern. Creating an index means that all occurrences of the pattern in the data graph are materialized. We envisage a syntax similar to SQL as the example below illustrates. The basic graph pattern enclosed in brackets describes the index criterion. The example below creates an index with the name I_1 on all subgraphs of the RDF graph `http://example.org/university.owl` having the form of the indexed pattern.

```
CREATE INDEX I1 ON <http://example.org/university.owl>
    ({?pub lub:publicationResearch ?res .
      ?grp lub:researchProject ?res .})
```

Before we can formally define indexes on RDF graphs, we first clarify the terms occurrence and solution of a basic graph pattern. Both definitions base on substitution functions mapping the variables of a graph pattern into the set of RDF terms and variables, i.e., $s : V_P \rightarrow \mathcal{V} \cup \text{RDF-T}$ where V_P denotes the variables contained in a pattern P. As an abbreviation, $s(P)$ denotes the result of applying s to P, i.e., replacing every occurrence of a variable $v \in V_P$ by $s(v)$.

Definition 2 introduces an important term for this paper: the occurrence of a pattern. It is essential to define the solution of a basic graph pattern and to describe the relationship between a query pattern and the indexed patterns.

Definition 2 (Occurrence of a pattern). *Let P_1 and P_2 be two basic graph pattern, P_1* occurs in *P_2 (up to blank node renaming), denoted by $P_1 \sqsubseteq P_2$, iff there exists a substitution function s such that $s(P_1) \subseteq P_2$.* □

If a pattern P_1 occurs in P_2 we refer to each $s(P_1)$ as an *occurrence* of P_1 in P_2. Furthermore, we denote by $S(P_1)$ the set of all occurrences of P_1 in P_2. If the meant pattern is clear from the context, we abbreviate $S(P), S(P_1), \ldots$ by $S, S_1, \ldots$, respectively, to improve readability.

Figure 2 provides an example for Defintion 2. P_1 occurs in P_2 with a substitution function that maps the variables `?pub` and `?person` of P_1 to `?paper` and `ex:bach` of P_2, respectively. This is the only occurrence of P_1 in P_2.

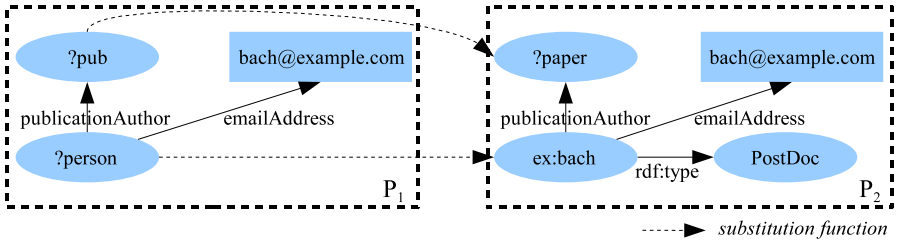

Fig. 2. The basic graph pattern P_1 occurs in P_2

A special case of Definition 2 represents the occurrence of a pattern P in an RDF graph G, because the range of the substitution function s does not contain any variables, i.e., $\forall v \in P : s(v) \notin \mathcal{V}$. If $s(P) \subseteq G$ holds then we call the substitution function s *solution* of P in G. As proposed in the SPARQL specification [12], we use simple entailment to define the solution of a basic graph pattern. This means, that no triple needs to be inferred or constructed. Determining a solution of a basic graph pattern is therefore equivalent to the subgraph isomorphism problem.

The following definition uses the previously introduced terminology to define indexes over RDF graphs.

Definition 3 (Index). *An* index *I over an RDF graph G is a pair $I = (P, S)$, where P is a basic graph pattern and S is the set of all solutions of P in G.* □

Subsequently, we refer to the basic graph pattern of an index as *index pattern*.

4 Index Selection

A query engine has the tasks to generate a set of query execution plans (QEPs), to estimate their costs, and to execute the cheapest one. In general, query processing is divided into four phases: query parsing, query rewriting, QEP generation and QEP execution. In this paper, we focus on QEP generation. This phase includes the selection of algorithms to implement each of the operators of the logical plan constructed by the previous phases. In order to minimize the costs of query execution, the query optimizer has to consider the existence of indexes or other access paths, distribution of stored data values, physical clustering of records, etc. [21]. **Question 1: Ist [21] eine schlechte Referenz?**

The selected access paths, e.g., scan (entry by entry) or access by an index, are an important component of the overall costs of a query plan. Most systems use a cost function that is basically an estimate of the number of disk I/O's involved, though some do take CPU utilization into account [21]. At the moment, we assume the existence of a cost function that assigns a cost value to a QEP and focus on the selection of indexes during the stage of query plan generation. In its basic form, the query optimizer selects a single index from a set of available indexes which is described in the first part of this section. Afterwards, we extend the basic problem to the selection of multiple indexes, e.g., selecting a subset of the available indexes.

Given a set of indexes and a query on an RDF graph, the query processor can only exploit some of the indexes to answer the query, the so-called *eligible indexes*. At the moment we require that the index pattern is completely contained in the query pattern. If an index covers the query only partially then the index may not contain all occurrences being in the result of the query and an expensive post processing step may be necessary. The following definition characterizes the set of indexes being of interest.

Definition 4 (Eligible index). *Let Q be a query and G be an RDF graph. An index $I = (P, S)$ is* eligible *for use in answering the query Q on G if I is an index over G and P occurs in Q, i.e., $P \sqsubseteq Q$.*

Example 1. We illustrate Definition 4 with an example. Let Q be a query and I_1 and I_2 be two indexes having the following pattern, respectively:

```
P(Q)  = {?postdoc lub:advisor ?advisor ;
                  lub:emailAddress ?email .
         ?pub lub:publicationAuthor ?postdoc .
         ?pub lub:publicationResearch ?res . }
P(I1) = {?pub lub:publicationResearch ?res .
         ?grp lub:researchProject ?res . }
P(I2) = {?postdoc lub:advisor ?advisor ;
                  lub:emailAddress ?email .
         ?pub lub:publicationAuthor ?postdoc . }
```

The index I_2 is eligible for answering the query Q, because the pattern of I_2 occurs in the query pattern. In contrast, the pattern of the index I_1 does not occur in in the pattern of Q, e.g., the triple pattern `(grp, lub:researchProject, ?res)` is not part of the query. Thus, I_1 is not eligible. Figure 3 visualizes the relationships between the patterns of Q, I_1, and I_2. □

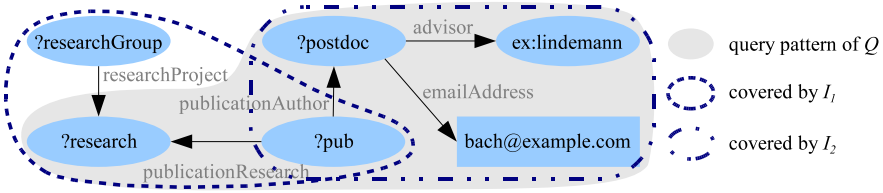

Fig. 3. Given a query Q, index I_1 is not eligible index while index I_2 is eligible

Definition 4 describes the set of indexes which may be used for query processing. However, some of the eligible indexes will reduce the costs of query evaluation more than other one. The query processor has the task to choose the best indexes according to a cost function. Formally, we could describe the task as follows: Let Q be a query, let $I_e = \{I_1, \ldots, I_n\}$ be the set of eligible indexes of Q, and let c be an arbitrary cost function. *Single index selection* chooses an index $I \in I_e$ such that the use of I minimizes the costs $c(Q)$ for executing the query Q.

Since the patterns of two eligible indexes may also overlap, a more interesting case arises if the optimizer chooses not only a single index but a bunch of possibly overlapping indexes for query evaluation. In the remainder of this section, we formalize the problem of selecting a subset from the set of eligible indexes. Before we can define the problem itself, we have to specify in which cases two indexes overlap.

We differentiate between intentionally and extensionally overlapping indexes. While the first type of overlapping is only defined on the index pattern, the second one analyzes the occurrences stored in the indexes. Definition 5 formally defines both types of overlapping.

Definition 5 (Overlapping indexes). *Let $I_1 = (P_1, S_1)$ and $I_2 = (P_2, S_2)$ be two indexes over an RDF graph G.*

- *I_1* intentionally overlaps *I_2 if there exists a substitution function s with $s(P_1) \cap P_2 \neq \emptyset$, i.e., $P_1 \sqsubseteq P_2$.*
- *I_1* extensionally overlaps *I_2 if $\forall s_1 \in S_1 \exists s_2 \in S_2 : s_1 \cap s_2 \neq \emptyset$.*
- *I_1 and I_2 intentionally/extensionally overlap if I_1 intentionally/extensionally overlap I_2 and vice versa.* □

Example 2. The two indexes I_1 and I_2 defined in Example 1 are not overlapping, because the index pattern do not share a triple pattern. If we added the triple

pattern `(?pub, lub:publicationAuthor, ?postdoc)` to I_1 then the indexes would intentionally overlap. □

The previous definition characterizes only the relationship between two indexes. If we consider a set of indexes it is too restrictive to require that any pair of indexes should overlap. Therefore, we define a set of overlapping indexes as follows:

Definition 6 (Set of overlapping indexes). *Let $\mathcal{I}$ be a set of indexes. A subset I_o of $\mathcal{I}$ is a* set of overlapping[1] indexes *if I_o satisfies the following properties $\forall I_1, I_2 \in I_o$:*

- I_1, I_2 *overlap* or
- $\exists I_{i_1}, \ldots, I_{i_k} \in I_o : I_1, I_{i_1}$ *overlap* $\wedge I_2, I_{i_k}$ *overlap* $\wedge I_{i_j}, I_{i_{j+1}}$ *overlap* $j = 1, \ldots, k-1$ □

Sets of overlapping indexes are good candidates for reducing query execution time, because the query engine can combine entries of these indexes to determine partial solutions of the query without matching the query pattern against the data graph. Sets of overlapping indexes are most beneficial if they are maximal. We say that a set of overlapping indexes is *maximal* if no index can be added such that the set is still overlapping, i.e., $\nexists I' \in \mathcal{I} : I_o \cup I'$ is a set of overlapping indexes. The property that indexes extensionally overlap is a stronger statement than that indexes intentionally overlap. Any pair of extensionally overlapping indexes are also intentionally overlapping, but the opposite is not true. Although deciding if two indexes extensionally overlap has a higher complexity, the estimation of their selectivity becomes more accurate (see Section 5.3).

Finally, we can define the main problem of our research: *index selection*. It is defined as follows:

Definition 7 (Index selection). *Let Q be a query, let I_e be the set of eligible indexes of Q, and let c be a cost function.* Index selection *chooses maximal sets of overlapping indexes from I_e such that the use of these indexes minimizes the costs $c(Q)$ for executing query Q.* □

5 Index Selection during Query Processing

While the previous sections focused on the formalization of the index selection problem, this section sketches the algorithm how the query optimizer can solve the index selection problem. Given a set of indexes $\mathcal{I}$ on an RDF graph G and a query Q, the algorithm consists of three steps:

1. Determine all eligible indexes $I_e \in \mathcal{I}$ of Q.
2. Determine all maximal sets of overlapping indexes I_o contained in I_e.
3. For each set $i \in I_o$ estimate its selectivity of the set i.

[1] intentionally or extensionally

In the first step, the algorithm determines the indexes that may be used for query evaluation. Since the maximal sets of overlapping indexes are good candidates to cover a large part of the query pattern, they are computed in the second step. Finally, the algorithm estimates the selectivities of index these sets, because they are an important indicator for the cost of query evaluation. In this section, we discuss each step separately.

5.1 Determining Eligible Indexes

The query optimizer has to determine the set of indexes that can potentially be used for evaluating a given query. According to Definition 4 the optimizer has to test for all available indexes if their index pattern occurs in the query pattern, i.e., check for each index if there exists a substitution function such that index pattern occurs in the query pattern. Note, that there may exist more than a single substitution function having this property, if the index pattern occurs at more than once in the query pattern. The problem is tightly connected to the problem of containment of relational (conjunctive) queries (see for instance [22]).

A simple algorithm to test for eligibility of an index starts with an empty set of substitution function and constructs, triple pattern by triple pattern, a mapping between the index pattern and the query pattern. The algorithm iterates over the triples of the index pattern, and incrementally constructs the substitution functions. If one of these triple patterns does not occur in the query pattern or lead to an inconsistent substitution function then the investigated index can definitely be marked as non-eligible. While generating the substitution function, the algorithm has to ensure that only consistent mappings are generated. For instance, there must not exists a mapping from a variable of an index pattern to two different resources or variables of the query pattern. The complexity of the algorithm is $O(\sum_{J \in I} |Q|^{|P_J|})$, where $|\cdot|$ denotes the number of triple patterns contained in a pattern and I denotes the set of all available indexes. To iterate over the triples of the query pattern is inefficient, because if a triple pattern of the query pattern does not occur in an index pattern, this index could still be eligible.

Alternatively, we can reformulate the problem of finding eligible indexes as a query. Hereby, we substitute the variables of the query pattern by new and unique resource identifiers and interpret it as an RDF graph. Furthermore, we interpret the index pattern as as SPARQL queries; if there exists a solution then the index is eligible. Since the query pattern and index pattern are generally small, this is an easy method to find eligible indexes. This approach is derived from the frozen facts algorithm used to solve the query containment problem [23].

Example 3. We use the query pattern and the index pattern of I_2 from Example 1 to illustrate the frozen fact algorithm. Each variable contained in the pattern of Q is substituted by a new resource identifier. The result of this transformation $s(Q)$ is interpreted as an RDF graph and the pattern of the index I_2 is matched against this graph (see Figure 4). In this example the matching result is not empty. Thus, the pattern of I_2 occurs in Q. □

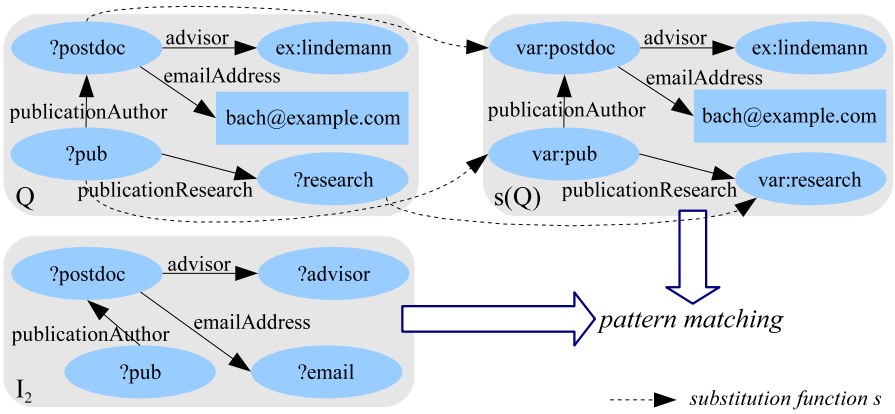

Fig. 4. Determining eligible indexes using the frozen facts algorithm

5.2 Determining Maximal Sets of Overlapping Indexes

In Section 4, we introduced sets of extensionally and intentionally overlapping indexes as candidates for reducing query execution time. The second step of the index selection algorithm determines the sets of overlapping indexes contained in the set of eligible indexes. Calculating the extensional overlap between two indexes is potentially expensive, because all occurrences of one index have to be compared with all occurrences of the other one. Thus, this step should not be executed during query compilation but is preferably computed at index creation time. Note, that a test for intentional overlaps (which is cheap to compute as only the triple patterns are concerned) can be used as an effective filter for testing extensional overlaps.

The problem of determining the maximal sets of indexes is usually solved by dynamic programming. Starting with a pair of indexes, the set is incrementally expanded by an index and it is checked if the set is still overlapping. Sophisticated heuristics would avoid to explore the complete search space. Instead of following this approach, we interpret the overlap relationship between indexes as an adjacency graph (see Definition 8) and exploit the fact that overlapping indexes belong to the same connected component in the adjacency graph (see Lemma 1).

Definition 8 (Adjacency Graph). *Let I be a set of indexes. A graph $G_A = (V, E)$, where V is the set of vertices and E is the set of edges, i.e., $E \subseteq V \times V$, is the* adjacency graph *of I if there exists a mapping $m : I \rightarrow V$ with the following properties:*

- *m is bijective*
- *Let $I_1, I_2 \in I$ be two indexes. $(m(I_1), m(I_2)) \in E \Leftrightarrow I_1, I_2$ overlap*[2] □

[2] intentionally or extensionally

Considering the adjacency graph of a set of indexes, we make the following important observation:

Lemma 1. *Let G_A be the adjacency graph of a set of indexes I. For any two indexes $I_1, I_2 \in I$ the following holds: I_1 and I_2 belong to the same connected component in G_A if and only if I_1 and I_2 overlap.*

Proof. Follows directly from Definition 8. □

Applying Lemma 1 to the index selection problem, we only need to determine the connected components of the adjacency graph corresponding to the set of eligible indexes to obtain the maximal sets of overlapping indexes. Knowing these sets is important for estimating their selectivity and, thus, the costs of a query execution plan.

5.3 Estimating Costs

Up to now we only assumed the existence of a cost function used to estimate the costs of a query execution plan. A widely accepted influence factor on the costs is the selectivity of an operation, i.e., the ratio between the size of the operation result and the size of the operation input. In this section, we first introduce the selectivity of a single index before we consider sets of indexes. Based on the results of this section, the plan optimizer can decide on the set of overlapping indexes to be used in query evaluation.

Given a query Q over an RDF graph G and an eligible index $I = (P, S)$, we have to calculate the ratio between the size of entries selected by an index and the size of the input data. Since the pattern P occurs $|S|$ times in the RDF graph G, an operator using the index I produces $|S|$ subgraphs of G as its result. The size of the input data is the number of all subgraphs in G having at most the size $|P|$, e.g., all potential solutions of the index pattern P. Note, that a single triple of an RDF graph can match more than one triple pattern of an index pattern. Therefore, we define selectivity of an index as follows:

Definition 9 (Selectivity of a single index). *Let G be an RDF graph and $I = (P, S)$ be an index over G. The selectivity of I, denoted as $sel(I)$, is defined as follows:*

$$sel(I) = \frac{|S|}{|G|^{|P|}}$$

□

A small selectivity value of an index is better, because only a small number of occurrences is obtained from the index and has to be considered in further query processing, e.g., testing for subgraph isomorphism.

We now discuss the selectivity of a set of indexes. In this case, selectivity generally decreases, because an occurrence of one index pattern can be combined with any occurrence of another one. Thus, the number of occurrences multiply. The selectivity of two indexes depends on the overlapping of the index pattern:

their index pattern may completely overlap, partially overlap, or not overlap at all. Therefore, we can derive the following estimation for the selectivity of a set of indexes:

Lemma 2 (Selectivity of a set of indexes). *Let G be an RDF graph and $\{I_1, \ldots, I_n\}$ with $I_i = (P_i, S_i), i = 1, \ldots, n$ be a set of indexes over G. The upper bound for the selectivity of using these indexes for query processing is as follows:*

$$sel(I_1, \ldots, I_n) \leq \frac{\prod_{i=1,\ldots,n} |S_i|}{|G|^{\max\{|P_1|,\ldots,|P_n|\}}}$$

Proof. As any occurrence of one index can be combined with any occurrence of another one we have $\text{sel}(I_1, \ldots, I_n) \leq \text{sel}(I_1) \cdots \text{sel}(I_n)$. We can simulate the set of indexes by a single index having the merge all index pattern as index pattern. We define the merge of index pattern, denoted by $\sqcup$, similar to the merge of RDF graphs. The size of the resulting index pattern is at least $|P_1 \sqcup \ldots \sqcup P_n| \geq \max\{|P_1|, \ldots, |P_n|\}$. Then we can estimate the selectivity of a set of indexes as follows:

$$\begin{aligned}\text{sel}(I_1) \cdots \text{sel}(I_n) &\leq \frac{\prod_{i=1,\ldots,n} |S_i|}{|G|^{\{|P_1 \sqcup \ldots \sqcup P_n|\}}} \\ &\leq \frac{\prod_{i=1,\ldots,n} |S_i|}{|G|^{\max\{|P_1|,\ldots,|P_n|\}}}\end{aligned}$$

□

Lemma 2 bases on the assumption that we do not know anything about the set of indexes. Additional information about the indexes allow for a more accurate estimation of their selectivity. For example, if we know that the index pattern are extensionally overlapping then the selectivity changes as stated in the following lemma.

Lemma 3. *Let G be an RDF graph and $\{I_1, \ldots, I_n\}$ with $I_i = (P_i, S_i), i = 1, \ldots, n$ be a set of indexes over G. If $\{I_1, \ldots, I_n\}$ is extensionally overlapping, then the following estimation of the selectivity of using these indexes for query processing holds:*

$$sel(I_1, \ldots, I_n) \leq \frac{\min(|S_1|, \ldots, |S_n|)}{|G|^{\max\{|P_1|,\ldots,|P_n|\}}}$$

Proof. Because the indexes are extensionally overlapping (see Definition 6), at most $\min(|S_1|, \ldots, |S_n|)$ occurrences can be selected by the set of indexes. □

Taking the selectivity of the eligible indexes into account the query optimizer can decide which indexes are valuable for query processing. Lemma 3 illustrates that maximal sets of overlapping indexes are a good choice of indexes to reduce query execution time.

6 Integration into a Query Processor

In [24] we describe the SPARQL query graph model (SGQM) which we use to represent a SPARQL query during query processing and to store additional information about the compilation process. The SQGM forms a key data structure for query processing. To validate our approach of rule-based rewriting of queries into queries that can be evaluated more efficiently, we implemented a query engine that is built on top of the Jena Framework [13]. After parsing a SPARQL query by ARQ, the query is translated into a SQGM and rewriting rules are applied. Subsequently, the query is transfered to the query execution engine of ARQ and evaluated (see Figure 5).

To integrate a support index selection into our query engine, an indexing component has to be implemented being responsible for managing indexes. Besides the usual tasks of an indexing component, e.g., create, delete, and update indexes, it has additionally the task to determine the overlaps between the indexes. As depicted in Figure 5 index selection is linked to two phases of the query processing: query rewriting and QEP generation. Beside others a goal of the query processor during the query rewriting phase is to reformulate the query such that available indexes can be used to answer the query, i.e., index pattern occur in the query. In the QEP generation phase the query processor constructs query execution plans and uses the selectivities of the eligible indexes to estimate the overall costs of the generated QEPs.

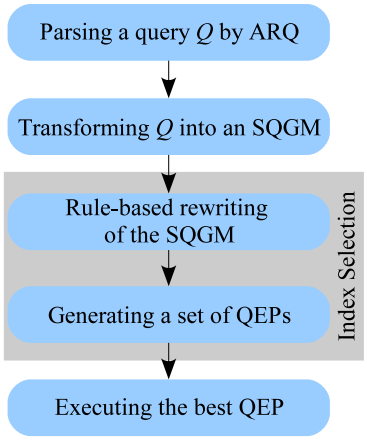

Fig. 5. SQGM based query processing of SPARQL queries

7 Conclusion

In this paper, we provided a framework for pattern-based indexing of RDF databases for SPARQL queries. Although there exist other RDF query languages similar to SPARQL, we chose SPARQL, because it will be the standard query language of the Semantic Web. However, our approach can also be applied to other RDF query languages, that bases on matching of triple pattern.

Similar to relational databases we suppose that users and administrators of an RDF database can define indexes on their RDF graphs. We define an index as a SPARQL query consisting of a single basic graph pattern whose occurrences in the underlying RDF database are computed in advance. How these occurrences are stored is a topic of future work.

We especially focused on the selection of not only a single index but a set of indexes. To decide on which indexes to use for query processing, we provided

the notion of selectivity of a set of indexes. The selectivity of an index is an important parameter to estimate the costs of an query execution plan involving this index. Future work includes the implementation of our approach and the evaluation of index selection during query processing.

Up to now we considered only queries consisting of a basic graph pattern. Our approach can be extended to `FILTER` clauses by including these clause in index patterns. Having `FILTER` clauses in the index pattern, determining eligible indexes becomes more complex, because the query processor has to check if query pattern is part of the index patterns. In future work, we will extend our approach to full SPARQL. For example, if we consider `OPTIONAL` clauses, then query rewriting becomes important for index selection. Since basic graph pattern can be grouped arbitrarily, the rewriting component is responsible to reformulate the query such that indexes can be used for query evaluation.

References

1. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Informatica **1** (1972) 173–189
2. Bentley, J.L.: Multidimensiuonal binary search in database applications. IEEE Transactions on Software Engineering **4**(5) (1979) 397–409
3. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGMOD Record, New York, NY, USA, ACM Press (1990) 322–331
4. Bertino, E.: Index configuration in object-oriented databases. The VLDB Journal **3**(3) (1994) 355–399
5. Beckett, D.: RDF/XML Syntax Specification (Revised). http://www.w3.org/TR/rdf-syntax-grammar/ (2004) W3C Recommendation.
6. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The RDFSuite: Managing Voluminous RDF Description Bases. In Decker, S., Fensel, D., Sheth, A.P., Staab, S., eds.: Proceedings of the Second International Workshop on the Semantic Web. (2001) 1–13
7. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF schema. In Horrocks, I., Hendler, J.A., eds.: Proceedings of the First International Semantic Web Conference. Volume 2342 of Lecture Notes in Computer Science., Springer (2002)
8. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2. In Cruz, I.F., Kashyap, V., Decker, S., Eckstein, R., eds.: Proceedings of the First International Workshop on Semantic Web and Databases. (2003)
9. Beckett, D.: The Design and Implementation of the Redland RDF Application Framework. In: Proceedings of the Tenth International Conference on World Wide Web, New York, NY, USA, ACM Press (2001)
10. Miller, L., Seaborne, A., Reggiori, A.: Three Implementations of SquishQL, a Simple RDF Query Language. Technical Report HPL-2002-110, HP Labs (2002)
11. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: Proceedings of the Eleventh International Conference on World Wide Web, New York, NY, USA, ACM Press (2002)

12. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/ (2006) W3C Candidate Recommendation.
13. Open Source: Jena – A Semantic Web Framework for Java. http://jena.sourceforge.net/ (2006)
14. Beckett, D.: Redland RDF Application Framework (2004) Supported by EU IST project SWAD-Europe.
15. Harth, A., Decker, S.: Optimized index structures for querying rdf from the web. In: Proceedings of the 3rd Latin American Web Congress, IEEE Press (2005)
16. Gutierrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of semantic web databases. In Deutsch, A., ed.: Proceedings of the 23st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, New York, NY, USA, ACM Press (2004) 95–106
17. Weigel, F., Schulz, K.U., Meuss, H. In: The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations. Volume 3671 of Lecture Notes in Computer Science. Springer Berlin/Heidelberg (2005) 49–67
18. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence **26**(10) (2004) 1367–1372
19. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In Weikum, G., König, A.C., Deßloch, S., eds.: Proceedings of the ACM SIGMOD International Conference on Management of Data, New York, NY, USA, ACM (2004) 335–346
20. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics **3**(2) (2005) 158–182
21. Date, C.J.: An introduction to database systems. 4th edn. Volume I. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
22. Lenzerini, M.: Data integration: a theoretical perspective. In Popa, L., ed.: Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, New York, NY, USA, ACM Press (2002) 233–246
23. Ramakrishnan, R., Sagiv, Y., Ullman, J.D., Vardi, M.Y.: Proof-tree transformation theorems and their applications. In: Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, New York, NY, USA, ACM Press (1989) 172–181
24. Heese, R.: Query graph model for sparql. In: International Workshop on Semantic Web Applications: Theory and Practice. Proceedings of ER workshops. (2006)