

The SPARQL Query Graph Model for Query Optimization

Olaf Hartig and Ralf Heese

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

Abstract. The Semantic Web community has proposed several query languages for RDF before the World Wide Web Consortium started to standardize SPARQL. Due to the declarative nature of the query language, a query engine should be responsible to choose an efficient evaluation strategy. Although all RDF repositories provide query capabilities, some of them require manual interaction to reduce query execution time by several orders of magnitude.

In this paper, we propose the SPARQL query graph model (SQGM) supporting all phases of query processing. On top of the SQGM we defined transformations rules to simplify and to rewrite a query. Based on these rules we developed heuristics to achieve an efficient query execution plan. Experiments illustrate the potential of our approach.

1 Introduction

With introducing the RDF data model researchers have investigated approaches to manage RDF data efficiently. As a part of these efforts, researchers as well as developers are looking for approaches to reduce query execution time. Current RDF repositories often rely on existing database technologies, e.g., relational databases [1–3] or Berkley-DB [4]. A main reason can be seen in the experience with efficient query processing gained over the past decades.

However, a posting in a newsgroup¹ illustrates that there is still room for improvement. A user of the Jena Semantic Web Framework [3] asked in a posting why his program containing only simple queries on a small RDF database runs so slowly. The answer was quite surprising: the user should put the more specific part of the query first, because it made a significant difference. Rearranging the queries resulted in a reduction of the execution time by a factor of 220, i.e., $33000ms \rightarrow 150ms$.

The development of RDF repositories came along with several proposals for query languages. Combining concepts of these languages, the World Wide Web Consortium currently standardizes a query language for RDF, namely SPARQL. Although the specification is still in the status of a working draft, it has already

¹ <http://groups.yahoo.com/group/jena-dev/message/21436> (posted on Mar 8, 2006)

been adopted by recent implementations of RDF repositories. Due to its declarative nature, a query engine has to choose an efficient way to evaluate a query. As shown in the above example, the user has still to choose the right order of triple patterns to minimize query execution time – contradicting to the nature of a declarative query language.

In this paper, we make a first step to consider all phases of query optimization in RDF repositories. We adopted the well-known query graph model developed for the Starburst database management system [5] to represent SPARQL queries and propose the SPARQL query graph model (SQGM). In our approach, this model forms the key data structure for all

phases of query processing and is used to store information about the query being processed. Figure 1 depicts the main phases of query processing in database systems. The small arrows depict the information flow and the large arrows depict the control flow. See [6] for a description of the phases. We defined transformation rules on top of the SQGM to provide means for rewriting and simplifying the query formulation.

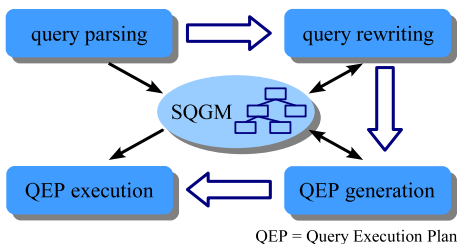


Fig. 1. Phases of the query processing

1.1 Running example

Throughout this paper we use the same SPARQL query (Figure 2) as a running example. The query asks for the names of all graduate students taking some course. Due to the `optional` clause in line 7, the result of this query may also include students taking no courses at all.

```

1 PREFIX ub: <http://www.lehigh.edu/.../univ-bench.owl#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 SELECT ?n ?c
4 FROM <http://example.org/University0.owl>
5 WHERE {
6     ?s rdf:type ub:GraduateStudent .
7     OPTIONAL { ?s ub:takesCourse ?c . }
8     ?s ub:name ?n .
9 }
```

Fig. 2. Example SPARQL query

Except for the prologue containing prefix declarations (lines 1–2), the structure of a SPARQL query is similar to the query language SQL for relational database systems. A SPARQL query consists basically of three parts: (a) the result specification part including solution modifiers (line 3), (b) the dataset definition part (line 4), and (c) the restriction definition part (lines 5–9). We refer to [7] for further explanation of the syntax .

1.2 Structure and Goals of this Paper

In the next section, we define the SPARQL query graph model and its graphical representation. Section 3 describes transformation rules based on this model which rewrite a query into a semantically equivalent one. The goal of rewriting a query is to achieve an efficient query execution plan. To evaluate our approach, we implemented it on top of the Jena Semantic Web Framework. We present and discuss some results of our experiments in Section 4. Section 5 discusses related work; Section 6 concludes the paper.

2 SPARQL Query Graph Model

In [5] Pirahesh et al. developed the *query graph model* (QGM) which defines a conceptually more manageable representation of an SQL query. We adapted this model to represent SPARQL queries – the SPARQL query graph model (SQGM). The basic elements of an SQGM are operators and dataflows – an operator processes data and a dataflow connects the output and input of two operators. In this section we describe the adaption of the query graph model to represent SPARQL queries. We begin with the description of the basic elements of the model and then explain the translation of SPARQL queries into the model.

2.1 Fundamentals

An SQGM can be interpreted as a directed labeled graph with vertices and edges representing operators and dataflows, respectively. Figure 3 shows the graphical representation of the SQGM for our example (cf. Figure 2). Operators are depicted as boxes consisting of a head, a body, and additional annotations. The Definitions 1 and 2 give a basic definition of an operator and a dataflow. In reference to the SPARQL specification, we refine the first definition and introduce operators having special properties and graphical representations below.

Definition 1. *An operator performs operations on its input data to generate output data.* □

An edge symbolizes the dataflow between two operators indicating that an operator consumes the output of another. Edges are directed and point to the data consumer.

Definition 2. *A dataflow connects two operators and transfers the data provided by one of them and consumed by the other.* □

An operator processes and generates either an RDF graph (a set of RDF triples), a set of variable bindings, or a Boolean value. Any operator has the properties *input* and *output*. The property *input* specifies the dataflow(s) providing the input data for an operator and *output* specifies the dataflow(s) pointing to another operator consuming the output data. We call the operator providing

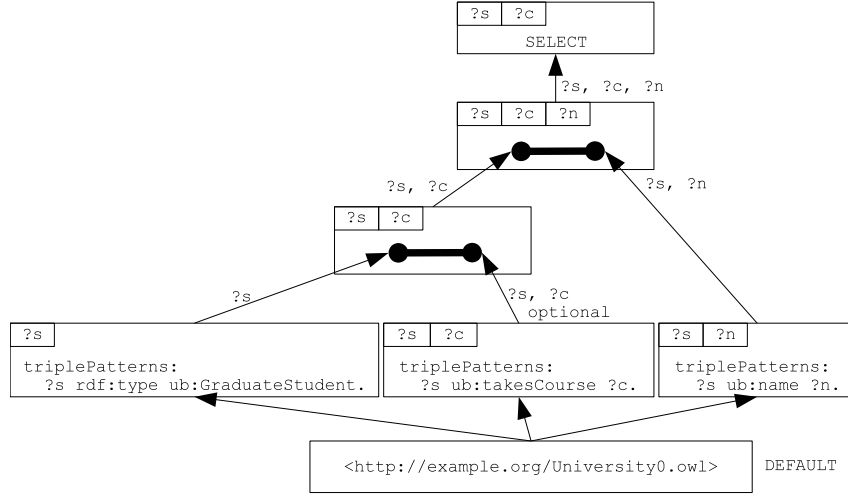


Fig. 3. Graphical representation of the SQGM for the SPARQL query in Figure 2

the data of a dataflow the *providing operator* of this dataflow. The operator consuming the data of a dataflow is called the *consuming operator*. The mapping $\text{UsingOp} : DF \rightarrow OP$ assigns the consuming operator to every dataflow and the mapping $\text{ProvOp} : DF \rightarrow OP$ assigns the providing operator to every dataflow.² The following expressions formally define these mappings; please note that we use a dot notation to access a property.

$$\begin{aligned} \text{UsingOp}(d) &:= o \Leftrightarrow d \in o.\text{input} \\ \text{ProvOp}(d) &:= o \Leftrightarrow d \in o.\text{output} \end{aligned}$$

According to the produced output of an operator we distinguish between *V-operator* and *G-operator*. A *V-operator* creates a set of variable bindings. The variables that are bound by an operator are specified in the property *provVars*. In the graphical representation, their names are listed in the head of the operator box. For example, in Figure 3 the right-most operator provides bindings for the variables *?s* and *?n*. The output of a *G-operator* is an RDF graph, i.e., a set of RDF triples. Since these operators do not bind any variables, they do not have the property *provVars* and the head is omitted in their graphical representation.

Dataflows are also divided into two categories: *V-dataflow* and *G-dataflow*. We denote with *GF* the set of all *G-dataflows* and with *VF* the set of all *V-dataflows*. *V-dataflows* are dataflows that originate in a *V-operator*, i.e., variable bindings are transferred. A *V-dataflow* has the property *vars* containing all variables that are used by subsequent operators. In the graphical representation a *V-dataflow* is annotated with the names of the variable contained in *vars*. It holds

² We denote with *OP* the set of all operators of an SQGM and with *DF* the set of all dataflows.

$\forall d \in VF : d.vars \subseteq ProvOp(d).provVars$ because not every consuming operator processes the bindings for all variables offered by the providing operator.

G-dataflows are dataflows that originate in a G-operator, i.e., an RDF graph is transferred. That is why they do not have the property *vars*.

Operator Types. We defined a set of operator types to cover the language structures of the SPARQL specification. Table 1 gives an overview of all defined operator types, their meaning, and their specific properties. If it is not noted otherwise, the values of the properties of an operator are listed in the body part of its box in the graphical representation.

Operator Type	Meaning and Properties
Graph Operator	Accesses an RDF graph <i>iri</i> : IRI of the RDF graph
Graph Merge Operator	Merges a set of RDF graphs
Graph Selection Operator	Accesses a set of named RDF graphs <i>var</i> : a variable bound to the IRI of the selected RDF graph
Graph Pattern Operator	<i>See detailed description below</i>
Join Operator	Joins two sets of variable bindings
Union Operator	Calculates the set union of two sets of variable bindings
Solution Modifier Operator	Applies solution modifiers to a set of variable bindings <i>distinct</i> : indicates duplicate elimination <i>orderBy</i> : determines the order of the result set <i>limit</i> : restricts the size of the provided set of variable bindings <i>offset</i> : an offset within the provided set of variable bindings
Select Result Operator	Returns only the bindings for the given variable names
Describe Result Operator	Creates an RDF graph describing a set of IRIs and the resources that are bound to given variable names <i>describedResources</i> : IRIs and variable names to be considered
Construct Result Operator	Creates an RDF graph by instantiating a template <i>template</i> : template graph pattern for constructing the query result
Ask Result Operator	Returns TRUE if the input is not empty

Table 1. Overview of all SQGM operator types

Due to the limited space we cannot define all operators in this paper. Instead, we selected one operator type, the graph pattern operator, which we describe in detail. The graph pattern operator corresponds to the basic graph pattern defined in the SPARQL specification. It is the main building block to specify the part of RDF dataset that is of interest. In the SPARQL query graph model the graph pattern operator is defined as follows:

Definition 3. A graph pattern operator is a V-operator which takes an RDF graph as input and returns the variable bindings for a set of triple patterns and value constraints as defined in [7]. The property input of a graph pattern operator

is restricted to G -dataflows. Furthermore, the following specific properties are defined:

- *triplePatterns*: a list of triple patterns to be matched
- *constraints*: value constraints to be satisfied by the variable bindings
- *contr*: indicates a provable contradiction in the value constraints

□

In the graphical representation of a graph pattern operator, the head of the box lists the variable names bound by the operator and the body contains the properties of the operator (see Figure 3).

SPARQL Query Graph Model. Before we describe the translation of a SPARQL query into an SQGM in the following section, we present the definition of SQGM.

Definition 4. A SPARQL query graph model (SQGM) represents a SPARQL query. It is a tuple $(OP, DF, r, dflt, NG)$ where

- OP denotes the set of all operators necessary to model the query,
- DF denotes the set of all dataflows necessary to model the query,
- r is an operator responsible for generating the result of the query ($r \in OP$),
- $dflt$ is an operator providing the default RDF graph of the queried RDF dataset ($dflt \in OP$),
- NG is the set of graph operators that provide the named graphs ($NG \subset OP$).

□

2.2 Translating a SPARQL Query to an SQGM

Our process for constructing an SQGM from a SPARQL query is described in Algorithm 1. It takes a query as input and returns the corresponding SQGM. The SPARQL query is given as a tuple (DS, GP, SM, R) where DS is the queried RDF dataset, GP is a graph pattern, SM is a set of solution modifiers, and R is the result form [7]. In the remainder of this section, we describe each step separately.

Algorithm 1 Translating a SPARQL query q into an SQGM Q

INPUT: $q := (DS, GP, SM, R)$ – a SPARQL query

OUTPUT: $Q := (OP, DF, r, dflt, NG)$ – an SQGM representing q

1. Generate operators for the RDF dataset DS ;
 2. Generate operators for the graph pattern GP ;
 3. Generate operators for the set of solution modifiers SM ;
 4. Generate operators for the result form R ;
-

Generate Operators for the RDF dataset. In the first step of the algorithm, the RDF dataset is modeled. An RDF dataset DS consists of a default RDF graph G and a set of named graphs $(\langle u_j \rangle, G_j)$. We use *graph operators* and *graph merge operators* to represent these parts. Although RDF graphs can be stored differently, e.g., in secondary storage or main memory, they are modeled in the same way. To model the RDF dataset the algorithm creates a graph operator for the default graph and each named graph (cf. Table 1). The property *iri* is set to the IRI of the respective RDF graph. The operators for the named graphs $(\langle u_j \rangle, G_j)$ are added to the sets OP and NG , the operator providing the default graph G is added to OP and is assigned to the *dflt* element. In the case that the default graph consists of a multiple RDF graphs, a graph merge operator is additionally created which provides access to the merge of these RDF graphs.

In the graphical presentation, the operator providing the default graph is additionally annotated with the keyword **DEFAULT**. In our running example (see Figure 3), the box at the bottom models the access to the default RDF graph.

Generate Operators for the Graph Pattern. The second step of query translation models the graph pattern GP of a SPARQL query. GP is either a basic graph pattern, a group graph pattern, value constraints, an optional graph pattern, an union graph pattern, or an RDF dataset graph pattern. Some of them (group graph pattern, optional graph pattern, union graph pattern, and RDF dataset graph pattern) contain other graph patterns. Thus, we use a tree of connected operators to represent GP and the algorithm traverses the graph pattern depth first to generate the operators. While traversing the graph pattern GP the algorithm creates the corresponding SQGM operators and dataflows bottom-up and adds them to the sets OP and DF , respectively. Furthermore, the properties of the operators and dataflows such as *input*, *output*, *provVars*, and *vars* are set accordingly.

In the following, we describe which operators are generated with respect to the different graph pattern types. For each *basic graph pattern* and *value constraint* a graph pattern operator is created and added to the set OP . Its properties *triplePatterns* and *constraints* are initialized according to the graph pattern at hand. Its property *provVars* contains all variables occurring in the basic graph pattern and value constraint. While constructing the SQGM the algorithm keeps track of the data source needed to set the *input* property of the new operators.

The example SQGM (Figure 3) contains three graph pattern operators – boxes containing a *triplePattern* property – representing the three basic graph patterns in our example query. The incoming edges represent G-dataflows indicating that these operators process the default RDF graph.

A *union graph pattern* $U(P_1, \dots, P_n)$ operates on a set of graph patterns P_i . It is modeled in two steps. First, the operators of all graph patterns P_i are created recursively. The result is a set of operator trees, one for each P_i . Second, the root operators of these trees are connected using union operators. Unions

involving three or more graph patterns are modeled as a binary tree of union operators.

A *group graph pattern* $G(P_1, \dots, P_n)$ containing multiple graph patterns P_i is modeled similarly to a union graph pattern. The only difference is that the root operators of the trees are connected by join operators. The example query in Figure 2 contains a group graph pattern consisting of two graph patterns: a optional graph pattern (line 6–7) and a basic graph pattern (line 8). It translates into a single join operators, i.e., the upper one. The second join operator is a result of translating the optional graph pattern which we discuss next.

An *optional graph patterns* $O(P_1, P_2)$ consists of two graph pattern P_1 and P_2 . It is basically modeled in the same way as a group graph pattern with two patterns, except that the dataflow between the operator representing P_2 and the join operator is initialized with the property *optional* set to TRUE. In the graphical representation this is reflected by the keyword `optional` (see Figure 3).

A *RDF dataset graph pattern* $\text{GRAPH}(g, P)$ matches a pattern P on one or more named graphs depending on whether g is an IRI or a variable. In case g is an IRI, the data source of the operators representing P is the graph operator that provides access to the RDF graph with the IRI g . Otherwise, if g is a variable, the algorithm creates a graph selection operator providing access to all named RDF graphs. Depending on its type the graph pattern P is modeled as described before.

Generate Operators for the Solution Modifiers. In the third step, the algorithm generates the operators representing the solution modifier set SM of the SPARQL query. Solution modifiers such as `order by` or `limit` manipulate their input data to change the order of the result set or to select a subset of it. If the set of solution modifiers SM of the SPARQL query is not empty, a solution modifier operator is created and its specific properties are initialized according to values in SM . The new operator is added to set OP of the SQGM and connected to the root operator representing the graph pattern GP .

Generate Operators for the Result Form. The last step of Algorithm 1 generates the operators for the result form R . The authors of the SPARQL specification [7] distinguish the four result forms `SELECT`, `DESCRIBE`, `CONSTRUCT`, and `ASK`. According to the result form of the query, the algorithm creates the appropriate operator and connects it to the SQGM generated so far, i.e., it constructs a dataflow to either the solution modifier operator or the operator representing the graph pattern. Furthermore, the operator specific properties are set. For example, if the result form is `CONSTRUCT`, then the template graph pattern is assigned to the property *template* of the construct result operator.

3 Query Rewriting based on SQGMs

While our query graph model for SPARQL is intended to support all phases of query processing (cf. Figure 1), we currently focus on query rewriting. In the

query rewriting phase, the generated SQGM is transformed into a semantically equivalent one to achieve a better execution strategy when processed by the plan optimizer. For instance, rules may aim at simplifying complexly formulated queries by merging graph patterns, e.g., avoiding join operations, and eliminating redundant or contradicting restrictions. In this section, we first define semantical equivalence of two SQGMs and, thereafter, specify transformation rules. These rules are finally combined to heuristics which promise to result in efficient query execution plans.

It is essential for query rewriting that applying transformation rules to a query has no impact on the query result. We define semantical equivalence of two SQGMs as follows:

Definition 5. *Two SQGMs q and q' are semantically equivalent, if the equation*

$$\text{Result}_D(q) = \text{Result}_D(q')$$

holds for any RDF dataset D , where $\text{Result}_D(q)$ denotes the result set of evaluating q on D . \square

We distinguish two categories of rules for SQGM restructuring: transformation rules and rewrite rules. While transformation rules change an SQGM only locally, i.e., an operator and its immediate neighbors are affected, rewrite rules are more complex and affect the complete SQGM. Since transformation rules are the building blocks of rewrite rules, we introduce them first.

Definition 6. *A transformation rule is a tuple (n, D, P, I) where n is the name of the rule, D is a set of operators for which the rule is applicable (domain), P is an optional set of Boolean valued expressions being the preconditions for applying the rule, and I is a non-empty list of instructions for changing the SQGM. \square*

Given a transformation rule, if an operator is contained in D and all preconditions P are fulfilled for this operator then the transformation rule can be applied to the SQGM, e.g., the instructions in I are executed. In the context of query rewriting, we are interested only in transformation rules which transform an SQGM into an semantical equivalent one. In the following, we give an example for the definition of a transformation rule.

Example 1. Figure 4(a) depicts a part of an SQGM. The two graph pattern operators can be merged with the join operator without affecting the semantics of the represented query. Merging would be beneficial in several cases, e.g., offering more options for index application or potentially facilitating further simplification. Therefore, we developed the transformation rule (MERGEJOINEDGPOS, D, P, I). It merges the join of two graph pattern operators to a single operator. The rule is applicable to join operators, i.e., $D = JO$, where JO denotes the set of all join operators.

Let $o \in D$ be a join operator, $(i_L, i_R) := o.\text{input}$ be the left and right input dataflows originating from the two operators $o_L := \text{ProvOp}(i_L)$ and

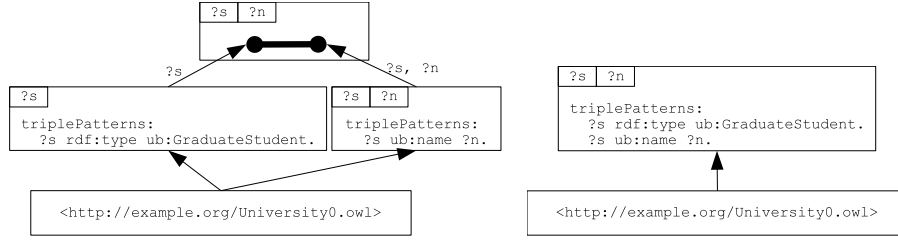


Fig. 4. Part of an SQGM before (a) and after (b) applying the transformation rule MERGEJOINEDGPOS

$o_R := ProvOp(i_R)$, and GPO be the set of all graph pattern operators in the SQGM. Then the set of preconditions P contains the following expressions:

- i) $o_L \in GPO$
- ii) $o_R \in GPO$
- iii) $i_L.optional = FALSE$
- iv) $i_R.optional = FALSE$
- v) $|o_L.output| = 1$
- vi) $|o_R.output| = 1$
- vii) $ProvOp(o_L.input) = ProvOp(o_R.input)$

Hence, the preconditions of MERGEJOINEDGPOS are the following: both operators are graph pattern operators (i and ii), none of them provides optional data (iii and iv), they are not used in another context (v and vi), and the input of both operators originates from the same RDF graph (vii).

The rule transforms $q = (OP, DF, r, dflt, NG)$ to $q' = (OP', DF', r, dflt, NG)$ using the following instructions in I :

- i) $o_L.triplePatterns := o_L.triplePatterns \sqcup o_R.triplePatterns$
- ii) $o_L.constraints := o_L.constraints \wedge o_R.constraints$
- iii) $o_L.contr := o_L.contr \vee o_R.contr$
- iv) $o_L.output := o.output$
- v) $o.output := ()$
- vi) $DF' := DF \setminus \{i_L, i_R, o_R.input\}$
- vii) $OP' := OP \setminus \{o, o_R\}$

The operator \sqcup used in the instructions i) denotes the merge of two triple patterns. This merge of triple patterns is similar to the merge of RDF graphs except that the triple patterns may contain variables. Especially, implementations of this operation have to consider the scope of blank nodes.

The operators in Figure 4(a) satisfy the preconditions of the transformation rule MERGEJOINEDGPOS. Thus, the rule can be applied to this part of the SQGM. Figure 4(b) shows the part after executing the instructions of the transformation rule. The three operators have been merged to a single graph pattern operator containing all triple patterns of the original graph pattern operators. \square

Rewrite rules are similar to transformation rules, but consider the complete SQGM. Every rewrite rule follows a certain *goal*, e.g., merge as many graph pattern operators as possible. To reach a goal it may be necessary to apply a single or a sequence of transformation rules several times. A goal of a rewrite rule is *reached* if no further steps are possible.

Definition 7. A rewrite rule is a tuple $(n, G, S, A_{check}, A_{compile})$ where n is the name of the rule, G specifies a goal, S defines steps to reach the goal, A_{check} is an algorithm to determine if another step is executable, and $A_{compile}$ is an algorithm that compiles a sequence of transformation rules to actually execute the next step. \square

In the case that multiple rewrite rules are applied to an SQGM, it may happen that the goal of a rewrite rule is contrary to the goal of another. Our current approach to solve this problem is to choose manually the set of rewrite rules to be applied.

Example 2. As mentioned in Example 1, merging joined graph pattern operators is beneficial in some cases. Following this assumption, we developed the rewrite rule MERGEALLJOINEDGPOS. Its goal is to merge as many graph pattern operators as possible.

In every step, the rewrite engine selects two graph pattern operators and merges them if possible. The algorithm A_{check} searches for a pair of candidate operators and checks if another step is executable. This is not trivial. As already mentioned, the translation algorithm constructs a join tree. Thus, graph pattern operators that could be merged by the transformation rule MERGEJOINEDGPOS, may occur at any place in the join tree. In order to merge these operators nevertheless, the join tree has to be restructured, so that the candidates become children of the same join operator. Due to the limited space, we do not discuss the transformation rules to restructure a join tree in this paper. However, the algorithm $A_{compile}$ compiles a sequence of these additional rules being suitable to restructure the join tree appropriate. After restructuring the transformation rule MERGEJOINEDGPOS finally performs the merge.

Considering Figure 3 as an example, the compiled sequence contains transformation rules to switch the places of the left-most and the right-most graph pattern operators and ends with the rule MERGEJOINEDGPOS. \square

Having rewrite rules defined, we are able to specify *heuristics*. A heuristic consists of a set of preconditions and a set of rewrite rules. If the preconditions are fulfilled and the rewrite rules are applied to an SQGM then the heuristic promises that in most cases the resulting SQGM meets a certain efficiency criterion, e.g., reducing query execution time. The following example illustrates the idea of one heuristic.

Example 3. We developed a heuristic that supports the fast path algorithm implemented in the Jena Semantic Web Framework [3]. The fast path algorithm detects triple patterns that can be executed as a single query within the underlying relational database system, e.g., the database can optimize the joins. These

triple patterns have to be part of the same execution stage, e.g., contained in the same basic graph pattern. We believe, that merging graph pattern operators of an SQGM reduces the query execution time, because larger sets of triple patterns are created and the fast path algorithm can push the evaluation of larger sets of triple patterns into the relational database. The heuristic suggests to apply rewrite rule MERGEALLJOINEDGPOS (cf. Example 2) if (a) the SQGM contains joined graph pattern operators, (b) the query RDF dataset is stored in a database, and (c) query execution is performed by Jena. \square

4 Implementation and Evaluation

We prototypically implemented the SPARQL query graph model on top of the Jena Semantic Web Framework and the ARQ query processor (Version 1.3). Afterwards, we run experiments to evaluate our approach. In this section, we outline our implementation, describe the testing environment, and present some results of our experiments.

The SPARQL query graph model is implemented as an extension to ARQ. While our query engine is derived from the classes of ARQ, the query model has been implemented from scratch. Additionally, we developed a rule engine being responsible for transforming query graph models.

Using our extension, a SPARQL query is processed as shown in Figure 5: First, the ARQ query processor parses the query and generates a query model specific to ARQ. We chose the indirection over the ARQ query model to reuse the SPARQL parser of ARQ. However, this model has the disadvantage that it is very close to the syntax of the query. For example: in contrast to ARQ, our model considers basic graph patterns and filter expressions as a single operation. Because the dataflows are not

explicitly defined in the ARQ model, it is not possible to distinguish between provided and actually used variables. After parsing the query, the generated query model is translated to an SQGM and heuristics are applied to provide a good basis for an efficient query execution plan. Then the restructured SQGM is translated back into an ARQ query model. ARQ generates a query execution plan which is executed finally.

Based on the Lehigh University Benchmark [8], we generated three sets of RDF data with increasing size using the scaling factors 1, 5, and 10, e.g., the sets contained about 100k, 624k, and 1272k triples, respectively. The data was managed by a relational database system and stored on secondary storage. Furthermore, we developed 41 queries which combined basic graph patterns, OP-

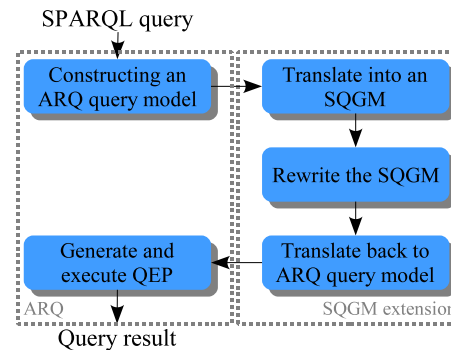


Fig. 5. Processing of a SPARQL query using the SQGM extension

TIONAL, FILTER and UNION clauses in various ways. For each query we measured the query execution time two times: with and without applying heuristics.

The diagram in Figure 6 illustrates the results of applying the heuristic presented in Example 3 to our example query. We see that the execution of the reformulated query is about 2.4 times faster than the original query. The reason for the better performance of our approach is that Jena implements the fast path algorithm. Considering our example, the algorithm can not facilitate combined pattern matching, because the correlation between the lines 6 and 8 in Figure 2 is not detected by Jena. The previously presented heuristic (cf. Example 3) transforms the query so that the fast path algorithm can facilitate to push down the pattern evaluation into the database.

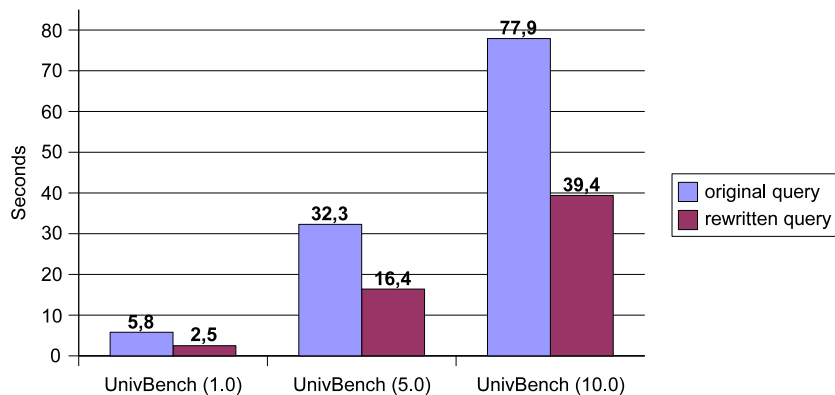


Fig. 6. Query execution time of our running example (Figure 2)

Further results of our experiments are the following. Parsing and transforming the query took less than 1 ms. Thus, these operations have little impact on the query execution time. The average savings were about 87%. Partly, this high savings are caused by a query which contained a contradicting value constraint. In contrast to our implementation, ARQ did not detect the contradiction. Instead of immediately returning an empty result, ARQ executed the query nevertheless.

5 Related Work

Although many approaches have been investigated how to store and to query RDF data, less attention has been paid to query optimization as a whole. However, some phases of query processing have already been considered. Cyganiak [9] and Frasinca et al. [10] considered query rewriting and proposed algebras for RDF. Derived from the relational algebra they allow to construct semantically equivalent queries. Furthermore, Serfiotis et al. developed algorithms for containment and the minimization of RDF/S query patterns in [11]. This approach examines only hierarchies of concepts and properties. All these approaches have

in common that they consider only a small part of the query processing, while the proposed query graph model supports all phases of query processing.

Pérez et al. present an approach to formalize the semantics of the core of SPARQL in [12]. They also define a set of equivalence expressions that allow to transform any SPARQL query into a simple normal form consisting of unions of graph patterns. Currently, we investigate how their formal model can be used to formulate transformation rules on top of SQGMs.

To improve the query execution time, several ways of storing RDF data have been developed and evaluated, e.g., Jena [3], Sesame [2], Redland [4] and a path-based relational RDF database [13]. But as the introductory example demonstrated, we have reasons to believe that developers of current RDF repositories have to re-engineer the query engines to declaratively deal with queries.

Christophides et al. focused on indexing RDF data, e.g., in [14] they developed labeling schemes to access subsumption hierarchies efficiently. Again, developing index algorithms is only a small part of query processing. It is also important to enable the query processor to transform the query such that an effective index can be invoked.

6 Conclusion and Future Work

Over the last years several approaches for storing and querying RDF data have been developed and evaluated. Although some research has been undertaken to provide means for query rewriting, we think that query optimization as a whole has not been considered so far. In this paper, we proposed the SPARQL query graph model (SQGM) supporting all phases of query processing. This model forms the key data structure for storing information that is relevant for query optimization and for transforming the query. We presented transformation rules which enable the query processor to transform an SQGM. We combined sets of transformation rules to rewrite rules as the base for heuristics. These heuristics enabled the Jena-based query execution to exploit the fast path algorithm more often. Our experiments demonstrated the potential of our approach to reduce query execution time.

The SPARQL query graph model can easily be extended to represent new concepts. This is important since the current SPARQL specification defines only basic query structures. Widely used structures such as group by, subqueries, and views are currently not supported, but will certainly be added in near future.

In our future work, we will develop further heuristics and exploit additional information to decide on the transformation rules being applied to the SQGM, e.g., information about the RDF schema or statistical data about the RDF datasets. Furthermore, we currently work on the problem of selecting indexes to minimize the costs of query execution. In the future, we want to combine the transformation of SQGMs with the selection of indexes. For example, apply rules to an SQGM such that an index with a high selectivity becomes usable. As a long-term goal, we will investigate extensions to the current SPARQL specification, e.g., subqueries and views, and develop appropriate transformation rules.

References

1. 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
2. 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)
3. 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)
4. 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)
5. Piraresh, H., Hellerstein, J.M., Hasan, W.: Extensible/rule based query rewrite optimization in Starburst. SIGMOD Records **21**(2) (1992) 39–48
6. Heese, R.: Query graph model for sparql. In: International Workshop on Semantic Web Applications: Theory and Practice. Proceedings of ER workshops. (2006)
7. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. <http://www.w3.org/TR/rdf-sparql-query/> (2006) W3C Candidate Recommendation.
8. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Journal of Web Semantics **3**(2) (2005) 158–182
9. Cyganiak, R.: A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Laboratories Bristol (2005)
10. Frasnar, F., Houben, G.J., Vdovjak, R., Barna, P.: RAL: an Algebra for Querying RDF. In Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E., eds.: Proceedings of the 13th International conference on World Wide Web, New York, NY, USA, ACM Press (2004)
11. Serfiotis, G., Koffina, I., Christophides, V., Tannen, V.: Containment and Minimization of RDF/S Query Patterns. In: International Semantic Web Conference. Lecture Notes in Computer Science, Springer (2005) 607–623
12. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L., eds.: Proceedings of the 5th International Semantic Web Conference. Volume 4273 of Lecture Notes in Computer Science., Springer (2006)
13. Matono, A., Amagasa, T., Yoshikawa, M., Uemura, S.: A path-based relational RDF database. In: CRPIT '39: Proceedings of the sixteenth Australasian conference on Database technologies, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2005) 95–103
14. Christophides, V., Karvounarakis, G., Scholl, D.P.M., Tourtounis, S.: Optimizing Taxonomic Semantic Web Queries Using Labeling Schemes. Web Semantics: Science, Services and Agents on the World Wide Web **1**(2) (2004) 207–228