A Cost Model for Querying Distributed RDF-Repositories with SPARQL

Philipp Obermeier and Lyndon Nixon

Freie Universität Berlin, Institut für Informatik, AG Netzbasierte Informationssysteme, Königin-Luise-Straße 24-26, D-14195 Berlin, Germany
obermeie@inf.fu-berlin.de, nixon@inf.fu-berlin.de

Abstract. In the last years, the query language SPARQL has evolved into the widely accepted standard for querying RDF. Since many Semantic Web applications make use of data whose storage and management is distributed, distributed SPARQL query processing becomes necessary. In the relation and object-oriented database community the efficiency gain by cost-based, adaptive optimizers for distributed querying is proven, though such optimizers are not available for SPARQL. Thus we describe in this paper a cost model which is meant to act as a sub component of a query optimizer for distributed SPARQL query processing to serve as a cost indicator for other subcomponents of the optimizer, e.g. query decomposition, query rewriting and choosing join algorithms and their order. The cost model is tailored for a heterogeneous grid of SPARQL processors and represents query plans as SPARQL Query Graph Models (SQGM). Costs are assigned in an System-R-like fashion relying on recursive cost and cardinality functions. Therefore evaluation complexities of basic operations in SPARQL queries are derived from the complexities of best practice algorithms for the algebraically equivalent basic operations in relational query languages.

1 Introduction

In January 2008 SPARQL was officially approved by a W3C Recommendation [15] – a status which no other RDF query language has achieved. Parallel to this development a respectable number of SPARQL query engines have been developed, e.g. Jena ARQ, the Sesame SPARQL plugin or OWLIM as part of ORDI.

Since many Semantic Web applications eo ipso manage and store data in distributed places, distributed SPARQL query processing becomes necessary. Development of distributed query engines is only in a premature state since none of the widely accepted query engines, including the ones mentioned above, are designed for distributed query processing at all. As we will point out in section 2 cost-based, adaptive optimization techniques are well established for querying distributed relational or object-oriented databases. Since SPARQL is based on the relational algebra of Relational Database Management Systems

(RDBMS), we believe it is possible to map these techniques to SPARQL. For this reason we present here a cost model for the evaluation of SPARQL queries tailored to act as part of an optimizer for adaptive, distributed query processing [4,11] supporting other subcomponents like query decomposition, query rewriting and join order selection for cost-based decision-making. Our cost model is similar to the ones for System-R suggested in [17,12] in that it estimates physical costs for each individual operation of a query and then totals these costs. Furthermore costs are distinguished by the type of used resources, i.e. number of CPU instructions, number of I/O operations (page/read accesses on persistent memory) and NET delay for retrieval of remote data. From this, a recursive cost function is developed for query execution based on physical cost functions and cardinality functions for basic operations in *SPARQL Abstract Queries* [15], a declarative representation of SPARQL queries (see section 3.1). Furthermore we presume a SPARQL query plan is given as a *SPARQL Query Graph Model* [9] which is a graph model for queries abstracting to the semantic of the corresponding SPARQL Abstract Query (see section 3.2). For the choice of complexities of basic SPARQL operations we argue that complexities of best-practice algorithms for related operations in the relational algebra for RDBMS can be used as orientation.

In Section 2 we address the scientific background of this paper. In Section 3 we introduce the preliminaries necessary for the definition of the cost and cardinality function in Section 4. There we also deliver the proof for the equality of the complexities for relational algebra operators and their equivalent SPARQL basic operations. In Section 5 we demonstrate the usage of the cost model as part of a query optimizer. We conclude with a discussion of open problems and future work.

2 Related Work

Some breakthroughs in the analysis of the expressiveness and classification of SPARQL also with respect to other query languages has been achieved by Perez et al.[13]. In their work they determined the complexity of certain types of SPARQL expression and developed an algebra for SPARQL resp. SPARQL graph pattern expressions which was a significant contribution to the theoretical algebraic model *SPARQL Abstract Queries* of SPARQL queries in [15]. Furthermore except for minor exceptions the possibility of a straightforward reduction of SPARQL to a SQL-resembling relational algebra has been shown in [3] which is the theoretical core of several SPARQL-to-SQL transforming SPARQL-DBMS like D2RQ or SquirrelRDF. Additional it can be attributed to the similarity between SPARQL and SQL algebras that a System-R-like graph model for SPARQL query plans together with optimization techniques query plans could be introduced [9].

Research in distributed SPARQL query processing started recently and predominantly leans on optimization techniques for general database query languages and RDF query languages. Heiner Stuckenschmidt et al. [18] suggest

for querying distributed RDF-repositories a schema-path-based indexing and storage organization as well as heuristics for join ordering. The techniques are implemented as a prototype extension for Sesame. In [2], *RDFPeers*, a spanning of RDF-Repositories over a Peer-to-Peer network is introduced. Execution of disjunctive and conjunctive queries over the distributed storages are described though RDFpeers doesn't support SPARQL as query language. In contrast to that Andreas Harth et al. [8] propose *YARS2*, a toolkit for efficiently querying distributed storages of graph-shaped data, especially RDF data and SPARQL are supported. A distributed indexing mechanism for quadruples (representing RDF triples) and distributed SPARQL query processing with join ordering are fundamental contributions of this work. While lookups for indexes can be dispatched concurrently in the distributed storage environment, other operations embodied in a SPARQL query (i.e. *Join*, *Union*, etc.) are solely executed at one host in the system. An adaptive cost estimation based on System-R-like physical cost and cardinality functions is presented in [16] for conjunctive reasoning for *Deductive Ontology Basis* (*DOB*), a subset of OWL lite. A translation of SPARQL queries into DOBs is promised by the authors for the future, but only partially explained at present.

In contrast to SPARQL the opportunities offered by distributed query processing are vastly explored in the database community. Kossmann describes in [11] a series of query processing optimization concepts in distributed databases in a textbook style. Pirahesh et al. [14] present rule-based rewriting techniques for queries which deliver optimization on a logical level independent from the physical implementation. Query plan cost models which sum up physical costs for single operators are suggested in [17,12]. Such a cost model is also needed as a subcomponent for an optimizer for distributed SPARQL processing as indicator for decision making of other components like query rewriting, decomposition or join and selection ordering. A sophisticated survey of adaptivity for query processing, especially intra-query adaptivity, was assembled by Deshpande et al. [4]. Adaptivity means the query processing is interleaved with adjustments by the query optimizer, which significantly improves the performance of the query evaluation

A System-R-like cost model and the cost-based, adaptive techniques addressed in the last paragraph are not available yet for a distributed SPARQL processing context. Thus our proposed cost model as part of a still to be developed optimizer for adaptive, distributed SPARQL query processing is a first step to change this circumstance.

3 Preliminaries

3.1 Formalization and Semantics of SPARQL Queries

3.1.1 SPARQL Abstract Queries For the study of SPARQL semantics, [15] provides a declarative formalization for a SPARQL query, called *SPARQL Abstract Query*. A SPARQL Abstract Query is a triple that consists of a *Query Form*, a *SPARQL Algebra Expression* and a *RDF Dataset*.

The set of Query Forms, denoted as $\mathcal{QF}$, comprises four functions $\{Select, Ask, Construct, Describe\}$ which take a *solution mapping multi-set* (see Definition 1) as input, *Construct* additional uses a set of tripe pattern templates. *Select* returns a set of variable bindings, *Ask* a boolean, *Construct* and *Describe* a RDF graph.

The SPARQL Algebra, whose terms are called SPARQL Algebra Expressions, is an algebra representing graph patterns and solution modifiers in queries. It provides a set of unary or binary functions $\{Join, Union, Diff, LeftJoin, ...\}$ and a set $\mathcal{BGP}$ of constants, which are called *Basic Graph Patterns (BGP)*. Both function and constants return a solution mapping multi-set as result while taking solution mapping multi-sets and logical boolean constraints for variable bindings as input. In the following we refer to the set of these functions as $\mathcal{SF}$, and we call $\mathcal{SAO} = \{\mathcal{BGP}\} \cup \mathcal{SF}$ the set of *SPARQL Algebra Operations (SAO)*. The elements of the subset $\mathcal{SM} \subset \mathcal{SF}$, where $\mathcal{SM} = \{Distinct, OrderBy, Limit, OffSet\}$, are called *Solution Modifiers (SM)*. Overall we refer to $\mathcal{QF} \cup \mathcal{SAO}$ as the set of *SPARQL basic operations*.

A *SPARQL Abstract Term* is either a SPARQL Abstract Query or a SPARQL Algebra Expressions. Intuitively a SPARQL Abstract Term describes a "sub term" of a SPARQL Abstract Query.

3.1.2 Solution Mapping Multi-Sets From a technical viewpoint solution mapping multi-sets are unordered lists of solution mappings which may contain duplicates. For our following discussion we adopt the mathematical definition from [15]:

Definition 1. *A* solution mapping multi-set *is a tuple* $(S, card)$ *where* S *is a set of solution mappings and* $card$ *is a function which annotates every* $x \in S$ *with a positive integer.*

The cardinality *of* $(S, card)$*, denoted as* $|(S, card)|$*, is the summation of* $card(x)$ *for all* $x \in S$*.*

3.2 SPARQL Query Graph Model

The *SPARQL Query Graph Model (SQGM)* [9] is a graph model for SPARQL queries resp. SPARQL Abstract Queries adopted from the *Query Graph Models* for SQL [14]. A SQGM is a planar rooted directed graph consisting of *operator* as nodes and *dataflows* as edges. Each operator represents an occurrence of an SPARQL algebra operation in the corresponding SPARQL Abstract Query while each data flow connects an operator A to an operator B, iff B uses A's result as input in the corresponding SPARQL Abstract Query. In this constellation A is called *providing operator* and B *consuming operator*. Furthermore operators are divided into types each of them having a one-to-one correspondence to a SPARQL basic operation, except for BGPs in combination with Filters and also for Solution Modifiers. For the first constellation SQGMs use an operator type, the set of *Graph Pattern Operators (GPO)*, each corresponding to a BGP with an optional Filter operation. This set is referred to as $\mathcal{GPO}$ in the following. For

Solution Modifiers SQGMs provide a general operator type, consisting of the *Solution Modifier Operators* which represent concatenations of arbitrary Solution Modifiers. Additionally we give the formal definition of a SQGM.

Definition 2. *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,*
- $r \in OP$ *is an operator responsible for generating the result of the query (i.e. r represents a Query Form),*
- $dflt \in OP$ *is an operator providing the default RDF graph of the queried RDF dataset,*
- $NG \subset OP$ *is the set of graph operators that provide named graphs.*

For particular details on SQGMs we refer to [9].

In the following we will base our cost estimation for SPARQL queries and query plans on their corresponding SQGMs. While as shown above queries can be translated directly into SQGMs query plans contain in general additional physical information for each operator. Thus we consider SQGM as abstraction of query plans in the sense that all physical information has been masked out.

3.2.1 Constraints for BGPs resp. Graph Pattern Operators Actually there are two different SPARQL basic operations which can describe joins between solution mapping multi-sets: The *Join* operator and BGPs containing more than one triple pattern. By limiting joins between solution mapping multi-sets to the actual *Join* operation of the SPARQL Algebra we unify these to one form to ease the definition of cost and cardinality functions. Motivated by this consideration we restrict BGPs (resp. GPOs) to the biggest subset of $\mathcal{BGP}$ (resp. $\mathcal{GPO}$), in which each BGP (resp. GPO) only contains exactly one single triple pattern. We refer to this subset as $\mathcal{BGP}_{TP}$ (resp. $\mathcal{GPO}_{TP}$). As shown in the extension to [13] it is semantically equivalent to simulate BGPs embodying an arbitrary set of triple pattern by defining a BGP for each triple pattern and after that to join these.

3.2.2 Constrains for Solution Modifier Operators As mentioned above a Solution Modifier Operator can express a application of several Solution Modifiers. In the following we only accept Solution Modifier Operators which represent exactly one Solution Modifier. A sequence of Solution Modifier appliances $m_1, \ldots, m_n$ in a SPARQL query can also be represented in a SQGM as a path of Solution Modifiers $M_1 \rightarrow \ldots \rightarrow M_n$ where M_i solely represents m_i. This limitation is motivated by the need of a one-to-one correspondence for the sake of exchangeability of cost and cardinality functions for corresponding SPARQL basic operations and SQGMs operator types. The exchangeability of functions significantly assists the comprehensibility of the cost computations based on SQGMs.

4 Cost and Cardinality Functions

In this section we will describe the structure of the cost model for SQGMs and provide a justification to base our cost estimation on the complexities of best-practice-algorithms for relational algebra operations related to the SPARQL basic operations.

Overall the cost model for SQGMs banks on a recursive cost function relying in turn on a recursive cardinality function. Furthermore both cost and cardinality function themselves are based on functions estimating physical cost and solution cardinalities for each SPARQL basic operation.

At first we introduce a method to assign cost and cardinality functions to each SPARQL basic operation also depending on the used implementation algorithms. After that we show how we can recursively compute costs for SQGMs based on our earlier results. Finally we justify that we can reduce almost all SPARQL basic operations to corresponding relational algebra operations in linear time to the input cardinalities.

4.1 Cost and Cardinality Functions for SPARQL Basic Operations

For each SPARQL basic operation we assign a *cardinality*, *CPU-* and *IO-cost function*. The cardinality function estimates the cardinality of the operation result, the CPU-function the necessary number of instructions and the IO-cost function the necessary number of hard disk (or page) accesses to process the operation. Each of these functions depends on the cardinalities of the input solution mapping multi-sets, the used algorithm and real number parameters expressing physical characteristics of the machine executing the query. To compute these functions for a triple pattern we use the pattern itself as indication for the cardinality of its solution mapping multi-set. This is motivated by the fact, that either we can retrieve the cardinality directly by sending the triple pattern to the RDF-Repositories or estimate the cardinality by a selectivity method with the triple pattern as input.

In Definition 3 we define a summarization of the function assignments above, denoted as *configuration*. Hereby $\mathcal{TP}$ stands for the set of all RDF Triple Pattern, $\mathcal{RDFG}$ for the set of all RDF-Graphs, $\mathcal{ALG}$ for the set of all algorithms implementing SPARQL basic operations. Moreover a *physical parameter setup* is defined as mapping from a set of identifiers to a set of reals. The set of all physical parameter setups is called $\mathcal{PPS}$. The Cartesian product $\mathbb{R}^{k_{op}}$ is the set of all possible tuples consisting of the cardinalities of the input solution mapping multi-sets for an operation *op*.

Definition 3. *A* configuration $\mathcal{C}$ *is a function assigning each SPARQL basic operation op a tuple* $\mathcal{C}(op) = (f_{CPU,op},\ f_{IO,op}, f_{card,op})$ *where*

if $op = \mathcal{BGP}_{\mathcal{TP}}$ *then* $f_{\phi,op} : (\mathcal{TP} \times \mathcal{RDFG} \times \mathcal{ALG} \times \mathcal{PPS}) \rightarrow \mathbb{R}_+$ *with* $\phi \in \{CPU, IO, card\}$,
else $f_{\phi,op} : (\mathbb{R}_+^{k_{op}} \times \mathcal{ALG} \times \mathcal{PPS}) \rightarrow \mathbb{R}_+$ *with* $\phi \in \{CPU, IO, card\}$.

We call $f_{IO,op}$ *resp.* $f_{CPU,op}$ *the* IO *resp.* CPU cost function *for op and* $f_{card,op}$ *the* cardinality function *for op.*

The IO- and CPU-costs of a SPARQL basic operation can be estimated accurately by the usage of an elaborated time and space complexity function for the implementation algorithm, given that the estimation errors for the cardinalities of the input solution mapping multi-sets and the physical parameters are small.

If no additional information is available, a naive way to compute the cardinalities of the results for a SPARQL basic operation is the usage of the result cardinalities provided by the operator definitions in [15]. For instance, we could use the possible maximum of the result size. Hence such a procedure, which solely relies on the input sizes of an operator, causes generally a large estimation error. If available, statistical values for the result sizes of Triple Patterns, BGPs or more complex SPARQL Algebra Expressions are a highly recommended alternative.

4.2 Cost and Cardinality Functions for SQGM

In this section we recursively determine the execution costs and result cardinalities of a query graph by beginning the recursion at its root moving to the nodes with in-degree 0. Relevant physical cost factors for the processing of an SQGM operator like costs for swapping, hashing, computation, etc. are given in a function, *physical parameter assignment*, and thus can be respected in the cost estimation. For the delay by transmission through the network we use a very simplistic approach by assigning a real number weight for each operator x which indicates the net distance between the host computing x and the host computing x's upstream operator. The actual net costs for x is the product <size of transmitted data> $*$ <distance weight>. To determine the overall net costs for a SQGM we add up these products for each operator.

First of all we emphasize in Definition 4 the exchangeability of cost and cardinality functions for SPARQL basic operations and SQGM operator types presuming the restrictions stated in Section 3.2.1 and 3.2.2.

Definition 4. *Given a SQGM* $(OP, DF, r, dflt, NG)$*, a configuration* $\mathcal{C}$ *and a SQGM operator type* t *which represents a SPARQL basic operation op with* $\mathcal{C}(op) = (f_{CPU,op},\ f_{IO,op}, f_{card,op})$*. We call* $f_{IO,op}$ *resp.* $f_{CPU,op}$ *the* IO *resp.* CPU cost function *for* t *and* $f_{card,op}$ *the* cardinality function *for* t*. Analogously we use for* t *the references* $f_{IO,t} = f_{IO,op}$*,* $f_{CPU,t} = f_{CPU,op}$ *and* $f_{card,t} = f_{card,op}$*.*

Before we can present the cardinality (Definition 5) and cost functions (Definition 6) for SQGMs, we need to determine three kinds of functions for a given SQGM $G = (OP, DF, r, dflt, NG)$:

An *algorithm assignment* is a function which assigns each operator $x \in OP$ an algorithm. This algorithm is a implementation of x according to the type x belongs to. For instance, if x belongs to the *Join Operator* type, the algorithm is a join algorithm (NestedLoop-Join, Hash-Join,...).

A *physical parameter assignment* is a function which assigns each operator $x \in OP$ a physical parameter setup. This physical parameter setup contains (approximation of) physical values relevant for the estimation of the processing costs (e.g. block size, costs for swapping, hashing or value comparison) for the machine executing x.

A *net-distance weight assignment* is a function which assigns each operator $x \in OP$ a positive real number. This number is a weight which expresses the net latency between the machine executing x and the machine executing x's upstream operator. The weight is multiplied with the result cardinality of x to take the net latency to the machine where the upstream operator is processed into account.

Finally we define the recursive cardinality and cost functions for SQGMs (Definition 5 and 6).

Definition 5. *For a SQGM $(OP, DF, r, dflt, NG)$, a configuration $\mathcal{C}$ and a node $x \in OP$, possessing the providing operators $c_1, ..., c_n$, the recursive function $card : OP \to \mathbb{R}_+$, called* cardinality function*, is defined as follows:*

if $x \in \mathcal{GPO}_{TP}$ then $card(x) = f_{card,\mathcal{GPO}_{TP}}(tp, G)$ where tp is the triple pattern embodied in x and G is the input graph of x,
else $card(x) = f_{card,x.type}(card(c_1), \ldots, card(c_n))$ where $x.type$ denotes the operator type of x.

Definition 6. *For a SQGM $G = (OP, DF, r, dflt, NG)$, $\phi \in \{IO, CPU, NET\}$, a configuration $\mathcal{C}$, an algorithm assignment $\mathcal{A}$, a physical parameter assignment $\mathcal{P}$, a net-distance weight assignment $\mathcal{D}$ and a node $x \in OP$ the recursive function $costs_\phi : OP \to \mathbb{R}_+$, called* $\boldsymbol{\phi}$-cost function*, is defined as follows:*

For $\phi \in \{IO, CPU\}$:
if $x \in \mathcal{GPO}_{TP}$ then $costs_\phi(x) = f_{\phi,\mathcal{GPO}_{TP}}(tp, G, \mathcal{A}(x), \mathcal{P}(x))$ where tp is the triple pattern embodied in x and G is the input graph of x,
else $costs_\phi(x) = f_{\phi,x.type}(card(c_1), \ldots, card(c_n), \mathcal{A}(x), \mathcal{P}(x)) + \; + \sum_{i \in \{1,\ldots,n\}} costs_\phi(c_i)$
where $x.type$ denotes the operator type, $c_1, ..., c_n$ the providing operators of x.
For $\phi = NET$:
$cost_{NET}(x) = \sum_{i \in \{1,\ldots,n\}} (\mathcal{D}(c_i) \cdot card(c_i) + costs_{NET}(c_i))$.

If x is a Select-, Describe-, Construct-, or Ask-Result Operator we refer to $costs_\phi(x)$ also as the $\boldsymbol{\phi}$-cost function for $\boldsymbol{G}$.

4.3 Cost Estimation Oriented on RDBMS Complexities

Based on [3] we will now prove that the choice of cost functions for SPARQL basic operations oriented on the complexities of best-practice-algorithms for relational algebra operations is sound. This is motivated by the fact that there

are strong similarities between SPARQL basic operations and the operations of the relational algebra for RDBMS. To support this claim we will prove that almost each SPARQL basic operation is reducible to a single operation or a more complex term of the relation algebra for RDBMS in linear time while the cardinalities of the input values are preserved.

Actually this reduction only yields an upper bound for SPARQL basic operations, which might be too pessimistic for certain implementations of operations. To formally neglect this we have to show that there also exists a reduction in the other direction. More precisely, each relational algebra operation must be reducible to a corresponding SPARQL Abstract Term (see Section 3.1), while the most efficient computation of the reduction function for this operation has to be in the smallest complexity class for which the evaluation of the SPARQL Abstract Term is a member of. Alternatively we could empirically prove the usability of the upper bound gained by the reduction introduced here. This proof, by mathematical means or benchmark testing, is a future task.

Before we present the reduction in Theorem 1, we want to clarify that we base the following discussion on the named version of the relational algebra, as described by [1] in chapter 5.1, which contains the operations $\{\sigma, \pi, \bowtie, \delta, \cup, -\}$. In addition, we define a database relation with duplicates as follows.

Definition 7. *A* relation with duplicates *is a tuple* $(r[U], card_{RA})$ *where* $r[U]$ *is a relation for a set of attributes* U *and* $card_{RA}$ *is a function which annotates every* $t \in r[U]$ *with a positive integer.*

The cardinality *of* $(r[U], card_{RA})$*, denoted as* $|(r[U], card_{RA})|_{RA}$*, is the summation of* $card_{RA}(t)$ *for all* $t \in r[U]$*.*

Consequently we extend the operations of the relational algebra to be capable of handling relations with duplicates as operands. For instance, the extended version of the relational algebra operation *Join*, called $\bowtie_{dup}$, additional takes care of the $card_{RA}$ values of its result tuples such that they resemble the *card* values of the SPARQL basic operation *Join*. We also apply this concept for the pairings ($\cup$,*Union*),($-$,*Diff*), (π,*Select*) and (σ,*Filter*)

Definition 8. *Let* A *be the set of all relations with duplicates. The functions* $\bowtie_{dup}$*,* $\cup_{dup}$*,* $-_{dup}$*,* π_{dup} *and* σ_{dup} *are defined as follows:*

Function $\bowtie_{dub}$*:* $A \times A \to A$ *is defined as* $\cup_{dub}((r_1, card_{RA,1}), (r_2, card_{RA,2})) = (r, card_{RA})$ *with* $r = r_1 \bowtie r_2$ *and for each* $t \in r$*,* $t_1 \in r_1$*,* $t_2 \in r_2$*,* $t = t_1 \cup t_2$ *holds* $card_{RA}(t) = card_{RA,1}(t_1) \cdot card_{RA,2}(t_2)$*.*

Function $\cup_{dub} : A \times A \to A$ *is defined as* $\cup_{dub}((r_1, card_{RA,1}), (r_2, card_{RA,2})) = (r, card_{RA})$ *with* $r = r_1 \cup r_2$ *and for each* $t \in r$*,* $t_1 \in r_1$*,* $t_2 \in r_2$*,* $t = t_1 = t_2$ *holds* $card_{RA}(t) = card_{RA,1}(t_1) + card_{RA,2}(t_2)$*.*

Function $-_{dub} : A \times A \to A$ *is defined as* $\cup_{dub}((r_1, card_{RA,1}), (r_2, card_{RA,2})) = (r, card_{RA})$ *with* $r = r_1 - r_2$ *and for each* $t \in r$*,* $t_1 \in r_1$*,* $t_2 \in r_2$*,* $t = t_1 = t_2$ *holds* $card_{RA}(t) = |card_{RA,1}(t_1) - card_{RA,2}(t_2)|$*.*

Function $\pi_{dub} : Att^n \times A \to A$ *is defined as* $\pi_{dub}(a, (r_1, card_{RA,1})) = (r, card_{RA})$ *with* $r = \pi(a, r_1)$*. For each* $t \in r$ *the cardinality* $card_{RA}(t)$ *is the summation*

of $card_{RA,1}(t_1)$ *for all* $t_1 \in r_1$ *with* $t = t_1|_t$. *Here* Att *is an arbitrary set of attributes and* Att^n *the n-ary Cartesian Product of Att,* $t_1|_t$ *is the tuple* t_1 *restricted to the attributes of* t.

Function $\sigma_{dub} : Att \times Dom \times A \to A$ *is defined as* $\cup_{dub}(a, d, (r_1, card_{RA,1})) = (r, card_{RA})$ *with* $r = \sigma(a, d, r_1)$ *and for each* $t \in r$, $t_1 \in r_1$, $t = t_1$ *holds* $card_{RA}(t) = card_{RA,1}(t_1)$. *Here Att is an arbitrary set of attributes and domains a set of arbitrary domains for a relational database.*

For the reduction in Theorem 1 we define in advance a simple helper algorithm, called *trans*, to transform a solution mapping multi-set $\mathcal{S} = (S, card)$ to an equivalent DB-relation with duplicates $\mathcal{R} = (r[U], card_{RA})$: The union of the variables in the domains of the solutions mappings in S is the set of attributes U for r. Thus the number of attributes in U is the overall number of different variables occurring in the the mappings in S. Furthermore each mapping $x \in S$ is represented by a tuple t_x in r with $card_{RA}(t_x) := card(x)$ where t_x has only values for the attributes which coincide with the domain of x (formally: attributes of t_x who don't represent a variable in x have a globally defined *null* value, which belongs to no domain of the database schema).

In the following we use the denotations: $trans(\mathcal{S}) = \mathcal{R}$, $trans(x) := t_x$, $trans(S) := r$, $trans(card) := card_{RA}$, $trans((S, card)) := (r, card_{RA})$ and $trans\text{-}att(S) = U$.
For this algorithm we imply (without explicit proof):

1. The number of attributes in U is the overall number of different variables occurring in the solution mappings in S
2. $x \in S$ iff $trans(x) \in trans(S)$
3. The algorithm's time and space complexity lies in $O(|(S, card)|)$.

Theorem 1. *Given a function f from SPARQL Abstract Terms to relational algebra terms with*

$$\begin{aligned} f(Join(\mathcal{S}_1, \mathcal{S}_2)) &= \bowtie_{dup} (trans(\mathcal{S}_1), trans(\mathcal{S}_2)), \\ f(Union(\mathcal{S}_1, \mathcal{S}_2)) &= \cup_{dub}(trans(\mathcal{S}_1), trans(\mathcal{S}_2)), \\ f(Diff(\mathcal{S}_1, \mathcal{S}_2)) &= -_{dub}(trans(\mathcal{S}_1), trans(\mathcal{S}_2)), \\ f(LeftJoin(\mathcal{S}_1, \mathcal{S}_2)) &= \cup_{dup}(\bowtie_{dup} (trans(\mathcal{S}_1), trans(\mathcal{S}_2)), \\ &\qquad -_{dup} (trans(\mathcal{S}_1), trans(\mathcal{S}_2))), \end{aligned}$$

where $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are two multi-sets of solution mappings. Then f is a reduction of the SPARQL basic operations Join, Union, Diff, LeftJoin to relational algebra terms, i.e., for* $\phi \in \{Join, Union, Diff, LeftJoin\}$ *the following statements hold*

- $x \in S$ *iff* $trans(x) \in r[U]$,
- $card(x) = card_{RA}(trans(x))$ *for each* $x \in S$,
- $trans\text{-}att(S) = U$,

where $\phi(\mathcal{S}_1, \mathcal{S}_2) = (S, card)$, $f(\phi(trans(\mathcal{S}_1), trans(\mathcal{S}_2))) = (r[U], card_{RA})$ *and both* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are two multi-sets of solution mappings. Moreover,* f *is computable linear in time with respect to the sum of cardinalities of* $\mathcal{S}_1$ *and* $\mathcal{S}_2$*, i.e.* $card(\mathcal{S}_1) + card(\mathcal{S}_2)$.

Proof. We show that for the operations *Join*,*Union*, *Diff* and *LeftJoin* the concluded statements hold.

A SPARQL Abstract Term $Join(\mathcal{S}_1, \mathcal{S}_2) = (S, card)$ with solution mappings multi-sets $\mathcal{S} = (S_1, f_1)$ and $\mathcal{S} = (S_2, f_2)$ is reduced by f to a (natural) join with duplicates in the relational algebra $\bowtie_{dup} (trans(\mathcal{S}_1), trans(\mathcal{S}_2)) = (r[U], card_{RA})$. For the generation of r from $trans(S_1)$ and $trans(S_2)$ operation $\bowtie_{dup}$ (Definition 8) solely relies on $\bowtie$. By the definition of $\bowtie$ ([1] p.58, top) and transformation algorithm *trans* it follows that $trans\text{-}att(S) = U$ and the fact, that a solution mapping x is element of S iff there exists a tuple t_x in $r[U]$, which has the same values for its attributes as x for its respectively variables. Moreover, algorithm *trans* uses for the cardinality $card_{RA}(r)$ of a tuple $t_x \in r[U]$ the same value as its corresponding solution mapping x (i.e. $trans(x) = t_x$). In addition, $\bowtie_{dup}$ (Definition 8) relies on the same computation formula for cardinalities of tuples as the SPARQL basic operation *Join* for solution mappings. Thus $card(x) = card_{RA}(trans(x))$ holds for each $x \in S$.

In an analogous way this can be shown for $Union$ and *Diff*. *LeftJoin* is defined as a SPARQL algebra term which combines *Join*, *Union* and *Diff*. Thus the reduction for *Join*, *Union* and *Diff* yields also a reduction for *LeftJoin*.

Furthermore, the definition of algorithm *trans* and reduction f implies that f is computable in linear time with respect to $card(\mathcal{S}_1) + card(\mathcal{S}_2)$.

For most of the SPARQL basic operations we can determine the complexity in the same way as in the proof of Theorem 1. Few are slightly different, e.g. $Filter$ allowing regular expression in SPARQL which occasionally can not be directly translated into the selection operation σ as pointed out in [3]. For these exceptions we will estimate their complexity based on known efficient implementations, also with orientation on similar operations in the relational algebra, if possible.

After the consideration above we are enabled to define a configuration oriented on the complexities for relational algebra operations for the example presented in Section 5. We use the complexity of relational algebra operations for the cost estimation of the related SPARQL algebra operations under the assumption that a SPARQL query processor is approximately as efficient as a RDBMS for the corresponding query (see Table 2). For SPARQL basic operations which are not reducible to relational algebra operations in the way described by Theorem 1, we rely on the complexities of best-practice implementations generally used by the database and Semantic Web community.

5 Example

At this point we showcase the expected benefits of a query optimizer based on our cost model. Therefore we require a system of networked, collaborating

Listing 1.1: SPARQL query q

```
PREFIX ub:<http://www.lehigh.edu/.../univbench.owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?n ? c
FROM <http://example.or/University0.owl>
WHERE {
    ?s rdf:type ub : GraduateStudent .
    OPTIONAL { ?s ub:takesCourse ?c . }
    ?s ub:name ?n .
}
```

hosts, whereas each host contains a RDF repository embodying such a query optimizer. We provide an exemplary SPARQL query q (Listing 1.1, Figure 1) and demonstrate how cost-based query decomposition will established an optimized distributed evaluation of the query in this system.

For this example we make the following presumptions for the host machines:

- The RAM size equals one block of persistent memory
- One I/O access equals reading of one block in persistent memory
- The RDF triples in the RDF repositories are indexed by a B+ tree as described in [7]
- Statistics about cardinalities of solution mapping multi-sets for SPARQL Algebra Expressions are available for each RDF repository (e.g. for BGPs a schema-path-based index of the result cardinalities for n-way triple pattern joins would be possible, analogous to the source index hierarchy proposed by [18])

In Table 2 we provide a list of IO and CPU cost functions necessary for this example, i.e. for each occurring operator type. These functions were derived from [6,5]. Table 1 explains the used symbols.

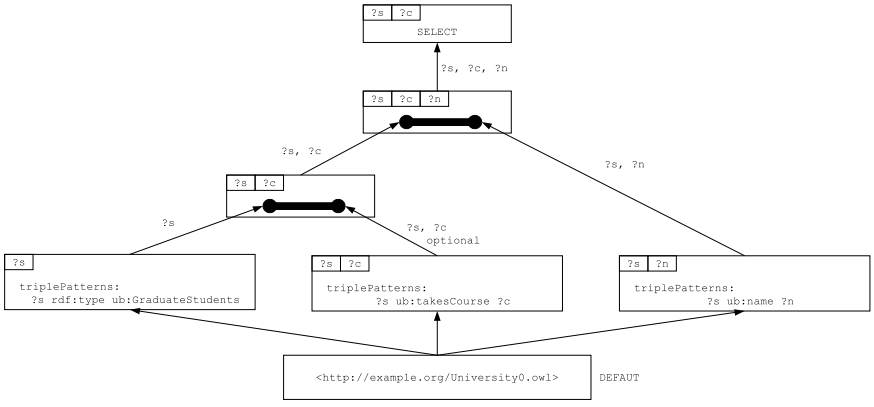

Fig. 1: Corresponding SQGM G for SPARQL query q

n_i	cardinality of i-th input solution mapping multi-set
b_i	blocks used by i-th input solution mapping multi-set
b_{res}	blocks used by the resulting solution mapping multi-set, i.e $\lceil n_{res}/b_{host} \rceil$
n_{res}	cardinality of the resulting solution mapping multi-set
l_i	average number of variables in i-th input solution mapping multi-set
$c_{generic}$	generic cost factor (if no other symbol is fitting)
$c_{compare}$	number of instructions to compare two variable bindings
c_{swap}	costs for swapping a block
c_{hash}	number of instructions to hash a key
n_{repo}	number of RDF triple stored in the RDF repository
b_{host}	used block size of the host, i.e. RDF triples per block

Table 1: Symbols used in the cost and cardinality formulas

Table 2: Cost functions for SQGM operator types

operator type	algorithm	cost function
Select Result Operator	full table scan	$f_{CPU} = n_1 \cdot c_{generic}$ $f_{IO} = b_1 \cdot c_{generic}$
Join Operator	NestedLoop-Join	$f_{CPU} = n_1 \cdot n_2 \cdot c_{compare}$ $f_{IO} = b_1 \cdot b_2 \cdot c_{swap}$
	Hash-Join	$f_{CPU} = min(n_1, n_2) \cdot c_{comp} +$ $+ max(n_1, n_2) \cdot c_{hash}$ $f_{IO} = (b_1 + b_2) \cdot c_{swap} + b_{res} \cdot c_{generic}$
	MergeSort-Join	$f_{CPU} = (n_1 \cdot logn_1 + n_2 \cdot logn_2 +$ $+ n_1 + n_2) \cdot c_{compare}$ $f_{IO} = (b_1 \cdot logb_1 + b_2 \cdot logb_2 +$ $+ b_1 + b_2) \cdot c_{swap}$
LeftJoin Operator	Union(HashJoin,Diff)	$f_{CPU} = f_{CPU,HashJoin}(n_1, n_2) +$ $+ (n_1 + n_2) \cdot c_{compare} +$ $+ n_1 \cdot c_{hash}$ $f_{IO} = f_{IO,HashJoin}(b_1, b_2) +$ $+ (b_1 + b_2) \cdot c_{swap} +$ $+ b_1 \cdot c_{generic}$
Graph Pattern Operator consisting only of a single triple pattern	B+-Tree lookup	$f_{CPU} = c_{comp} \cdot log_{b_{host}} n_{repo}$ $f_{IO} = c_{swap} \cdot (log_{b_{host}} n_{repo} + b_{res})$

Now we give a step by step walk-through of the distributed querying process:

1. Initially SPARQL query q (Listing 1.1) is transformed into a SQGM G (Figure 1).
2. Then the decomposition component determines – by calling the cost estimation component – the CPU-costs and IO-costs for each operator and thereby for each sub graph in SQGM. We use the following randomly chosen configuration of physical parameters for each operator in the SQGM G: $c_{generic} = c_{comp} = 1$, $c_{swap} = 10$, $c_{hash} = 2$, $n_{repo} = 10^6$, $b_{host} = 10$
   As implementation for the uppermost *Join* operator in G we use the MergeSort-Join algorithm (Note: A SQGM can be interpreted as rooted directed tree, if the Default Graph operator is ignored.). For all other operators in G the choice of the implementation is unambiguous (see Table 2).
3. After the decomposition component gets back the cost estimations (Figure 2) it decides the partitioning of SQGM G with one of the following strategies (These strategies are intuitive choices, their sophisticated research is targeted for the future.): If the query optimizer is regularly noticed about available resources, the partitions can be chosen w.r.t. to their costs and these resources, i.e. a matching between costs and resources is approximated. Alternatively the decomposition component tries to partition SQGM G into chunks with circa the same costs. Here we use the second strategy.
   The overall costs for the left and right sub-tree at the upper-most *Join* operator are $1012 + 1320 = 2332$ and $6 + 310 = 316$, if we weight CPU and

IO costs with factor 1. Assume two remote hosts A and B are available with access to q's default graph. A should have net-distance 3 and B net-distance 10 to the host which will process the uppermost $Join$ operator and the Select operator. The remote processing costs for the left subtree on host A would than be 2332 plus the net latency $3 \times 200 = 600$, for the right subtree on host B 316 plus the net latency $10 \times 250 = 2500$. Since A and B work parallel the overall cost is 2932 (if we have to wait until A is finished) for the remote processing of the both sub trees compared to a sequential local processing which is $2932 + 2816 = 6748$. Thus we split G at the uppermost-most $Join$ operator.

4. The resulting partitions for the sub trees of the uppermost $Join$ operator in G lead to the sub queries q_1 and q_2 (Listings 1.2 and 1.3), which then are forwarded to host A and B for processing. The retrieved sub query answers routed back are combined to an answer for q according to its decomposition.

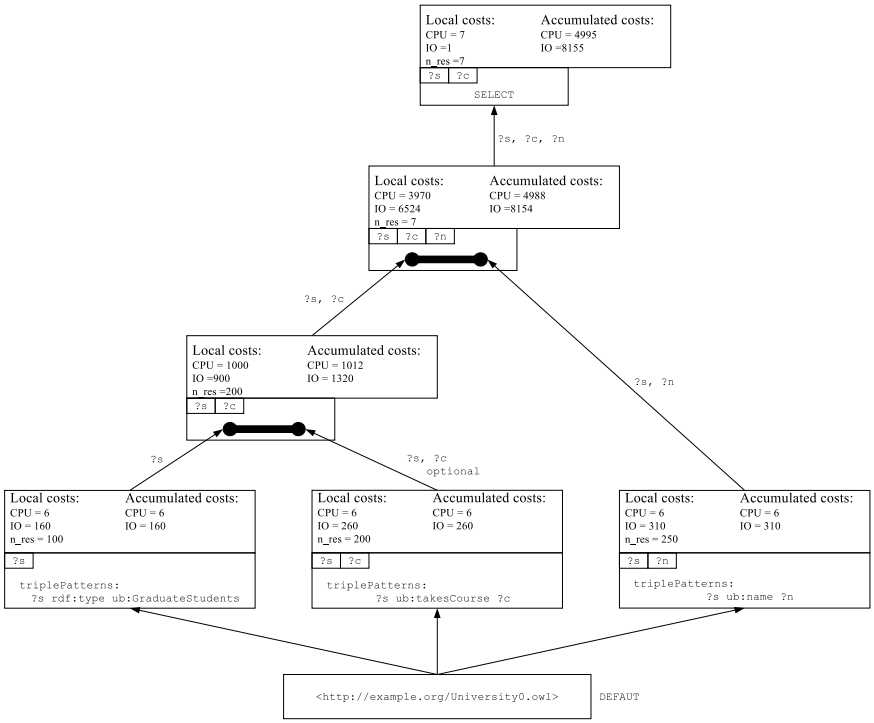

Fig. 2: Cost annotated SQGM G

Listing 1.2: SPARQL query $q1$

```
PREFIX ub:<http://www.lehigh.edu/.../univbench.owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?s ? c
FROM <http://example.or/University0.owl>
WHERE {
    ?s rdf:type ub : GraduateStudent .
    OPTIONAL { ?s ub:takesCourse ?c . }
}
```

Listing 1.3: SPARQL query $q2$

```
PREFIX ub:<http://www.lehigh.edu/.../univbench.owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?s ?n
FROM <http://example.or/University0.owl>
WHERE {
    ?s ub:name ?n .
}
```

6 Conclusion and Future Work

We have developed a cost model that provides System-R-like cost and cardinality functions for SPARQL queries given as SQGMs and evaluated in an distributed environment. In addition we proved by reduction that for the cost estimation of most SPARQL basic operations complexities of relational algebra operations can be used as upper bounds. Hence, it has yet to be proven – mathematically or empirically – that these upper bounds are not to pessimistic.

For the future we plan an implementation of the cost model as part of query optimizer for distributed query processing. Since the optimize-then-execute paradigm is not efficient for complex queries due to exponential growth of the estimation error [10] we additionally expect that the query processor and its optimizer operates in an *intra-query adaptive* fashion [4], i.e. the optimizer interleave query processing for dynamic adjustments and cost estimations to provide better response time or more efficient usage of resources.

We will evaluate the cost model in form of representative benchmarks tests for this implementation. In particular the accuracy of the upper bounds for SPARQL basic operations provided by the reduction mentioned above will be verified.

7 Acknowledgements

This work is partially supported by the TripCom (IST-4-027324-STP) project; `http://www.tripcom.org`.

References

1. Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases.* Addison-Wesley, Reading, Massachusetts, 1995.

2. Cai, M., and Frank, M. Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network. In *WWW '04: Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, 2004), ACM, pp. 650–657.
3. Cyganiak, R. A relational algebra for SPARQL. Tech. Rep. HPL-2005-170, Hewlett Packard Laboratories, Sept. 28 2005.
4. Deshpande, A., Ives, Z. G., and Raman, V. Adaptive query processing. *Foundations and Trends in Databases 1*, 1 (2007), 1–140.
5. Elmasri, R., and Navathe, S. B., Eds. *Fundamentals of Database Systems*, second ed. Benjamin/Cummings, 1994.
6. Graefe, G. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2 (June 1993), 73–170.
7. Harth, A., and 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, 2005), IEEE Computer Society, p. 71.
8. Harth, A., Umbrich, J., Hogan, A., and Decker, S. Yars2: A federated repository for querying graph structured data from the web. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea* (Berlin, Heidelberg, November 2007), K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. J. B. Nixon, J. Golbeck, P. Mika, D. Maynard, G. Schreiber, and P. Cudré-Mauroux, Eds., vol. 4825 of *LNCS*, Springer Verlag, pp. 211–224.
9. Hartig, O., and Heese, R. The SPARQL query graph model for query optimization. In *ESWC* (2007), E. Franconi, M. Kifer, and W. May, Eds., vol. 4519 of *Lecture Notes in Computer Science*, Springer, pp. 564–578.
10. Ioannidis, Y., and Christodoulakis, S. On the propagation of errors in the size of join results. In *19 ACM SIGMOD Conf. on the Management of Data, Boulder* (May 1991).
11. Kossmann, D. The state of the art in distributed query processing. *ACM Comput. Surv 32*, 4 (2000), 422–469.
12. Mackert, L. F., and Lohman, G. M. R* optimizer validation and performance evaluation for distributed queries. In *Twelfth International Conference on Very Large Data Bases* (Kyoto, Japan, 25–28 Aug. 1986), W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds., Morgan Kaufmann, pp. 149–159.
13. Perez, J., Arenas, M., and Gutierrez, C. The semantics and complexity of SPARQL.
14. Pirahesh, H., Hellerstein, J. M., and Hasan, W. Extensible/rule based query rewrite optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–5, 1992* (pub-ACM:adr, 1992), M. Stonebraker, Ed., vol. 21(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, ACM Press, pp. 39–48.
15. Prud'hommeaux, E., and Seaborne, A. SPARQL query language for RDF. W3C recommendation, W3C, Jan. 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.
16. Ruckhaus, E., Ruiz, E., and Vidal, M.-E. Query evaluation and optimization in the semantic web, 2007.
17. Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. Access path election in a relational database management system. *Proc. ACM-SIGMOD International Conference on Management of Data* (May 30 - June 1, 1979), 23–34.
18. Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., and Broekstra, J. Index structures and algorithms for querying distributed RDF repositories. In *WWW* (2004), S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, Eds., ACM, pp. 631–639.