SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation

Markus Stocker
HP Laboratories
Bristol
United Kingdom
markus.stocker@gmail.com

Andy Seaborne
HP Laboratories
Bristol
United Kingdom
andy.seaborne@hp.com

Abraham Bernstein
Department of Informatics
University of Zurich
Switzerland
bernstein@ifi.uzh.ch

Christoph Kiefer
Department of Informatics
University of Zurich
Switzerland
kiefer@ifi.uzh.ch

Dave Reynolds
HP Laboratories
Bristol
United Kingdom
dave.reynolds@hp.com

ABSTRACT

In this paper, we formalize the problem of *Basic Graph Pattern* (BGP) optimization for SPARQL queries and *main memory* graph implementations of RDF data. We define and analyze the characteristics of heuristics for selectivity-based static BGP optimization. The heuristics range from simple triple pattern variable counting to more sophisticated selectivity estimation techniques. Customized summary statistics for RDF data enable the selectivity estimation of *joined* triple patterns and the development of efficient heuristics. Using the Lehigh University Benchmark (LUBM), we evaluate the performance of the heuristics for the queries provided by the LUBM and discuss some of them in more details.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Algorithms, Performance

Keywords

SPARQL, query optimization, selectivity estimation

1. INTRODUCTION

In this paper, we focus on selectivity-based static *Basic Graph Pattern* (BGP) optimization for SPARQL queries [14] and *main memory* graph implementations of RDF [9] data. In SPARQL, a BGP is a set of triple patterns where a triple pattern is a structure of three components which may be concrete (i.e. bound) or variable (i.e. unbound). The three components which form a triple pattern are respectively called the subject, the predicate and the object of a triple pattern. Sets of triple patterns, i.e. Basic Graph Patterns, are fundamental to SPARQL queries as they specify the access to the RDF data.

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

Query optimization is a fundamental and crucial subtask of query execution in database management systems. We focus on *static* query optimization, i.e. a join order optimization of triple patterns performed before query evaluation. The optimization goal is to find the execution plan which is expected to return the result set fastest without actually executing the query or subparts. This is typically solved by means of heuristics and summaries for statistics about the data.

The problem we are going to tackle in this paper is best explained by a simple example. Consider the BGP displayed in Listing 1 which represents a BGP of a SPARQL query executed over RDF data describing the university domain. Typically, there are a number of different subjects working, teaching, and studying at a university (e.g. staff members, professors, graduate, and undergraduate students). They are all of *type* Person in our RDF dataset. We know that the dataset contains a huge number of RDF resources of type Person among others of type Publication, Course, Room.

The OWL [1] schema ontology used to describe the vocabulary for the RDF dataset states that the property for the social security number is inverse functional. Therefore, the object of the property uniquely determines the subject. Hence, the second triple pattern in our BGP of Listing 1 matches only one subject with the social security number "555-05-7880". Our schema ontology specifies further that the domain of the social security number property is a class of type Person. Therefore, we can state that the subject with social security number "555-05-7880" is of type Person (or our data is inconsistent).

The question is in which order a query engine should execute the two triple patterns. Given the research on join order strategies that has been pursued for relational database systems, we can safely state that a query engine should execute first the second triple pattern as its result set is considerably smaller compared to the result set of the first triple pattern. Therefore, a static optimizer should reverse the triple patterns. The join over the subject variable will be less expensive and the optimization eventually lead to better query performance. Note that, as the ontology schema specifies that the domain of the property for the social security number is a class *Person*, and provided that the data is

Listing 1: Example BGP

```
?x rdf:type uv:Person .
?x uv:hasSocialSecurityNumber "555-05-7880"
```

consistent, a static optimizer may even drop the first triple pattern as, in our example, a subject with a social security number has to be of type Person.

The main contributions of this paper are (1) a framework for static optimization of Basic Graph Patterns, (2) a set of heuristics for the selectivity estimation of joined triple patterns, (3) a proposal for summary statistics of RDF data to support heuristics in their selectivity estimation, and (4), a query performance evaluation for the heuristics that underlines the importance of query optimization for RDF query engines.

The focus in our work is on *main memory graph implementations* of RDF data (i.e. *in-memory* models). Currently most RDF toolkits support both in-memory and on-disk models. Relational database management systems (RDBMS) are commonly used as persistent triple stores for on-disk models. Because of the fundamentally different architectures of in-memory and on-disk models, the considerations regarding query optimization are very different. Whereas query engines for in-memory models are native and, thus, require native optimization techniques, for triple stores with RDBMS back-end, SPARQL queries are translated into SQL queries which are optimized by the RDBMS. It is not our goal in this paper to analyze optimization techniques for on-disk models and, hence, we are not going to compare in-memory and on-disk models. Furthermore, we focus on the *evaluation of the presented optimization techniques* without comparing the figures with the performance of alternative implementations. A comparison of implementations requires a comprehensive study that goes beyond the scope of this paper. In fact, the query performance of query engines is not just affected by static query optimization techniques but, for instance, also by the design of index structures or the accuracy of statistical information. Finally, our focus is on *static query optimization techniques*. Hence, we do not discuss optimal index structures for RDF triple stores, neither in-memory nor on-disk, as this too is a research topic that goes beyond the scope of this paper.

Our focus on main memory graph implementations, i.e. in-memory models, has an important limitation: scaling. Indeed, the few gigabytes of main memory clearly limit the size of RDF data which may be processed in main memory. Therefore, we might question the relevance of studying optimization techniques for RDF in-memory models. We argue, that in-memory models are important for a number of reasons. First, optimized queries on in-memory models run much faster than on-disk. Second, 64-bit architectures pose virtually no more limits to the theoretical amount of main memory in computers. Third, in a cluster, distributed in-memory models could be used for parallel query evaluation. Finally, optimization techniques and customized summary statistics of RDF data are important for native RDF persistent stores as they do not rely on relational database technology and, hence, require a native optimizer.

We believe, native optimization techniques and optimized summary statistics of RDF data are a key requirement for efficient SPARQL query evaluation on the Semantic Web.

The paper is organized as follows. In Section 2, we succinctly discuss related work and set our work in context. In Section 3, we present the theoretical background, the architecture, and the proposed heuristics implemented for the Jena ARQ [5] optimizer. In Section 4, we discuss our approach for summary statistics of RDF data, i.e. meta information about RDF data used for selectivity estimation of joined triple patterns. Section 5 describes our approach for selectivity estimation of (joined) triple patterns. Finally, in Section 6, we present the query performance evaluation we conducted for the optimizer and the proposed heuristics. We close the paper with future work and limitations.

2. RELATED WORK

The execution time of queries is heavily influenced by the number of joins necessary to find the results of the query. Therefore, the goal of query optimization is (among other things) to reduce the number of joins required to evaluate a query. Such optimizations typically focus on histogram-based selectivity estimation of query conditions.

Piatetsky *et al.* introduce in [12] the concept of selectivity estimation of a condition. In [15] Selinger *et al.* present the System R optimizer, a dynamic programming algorithm for the optimization of joins. Likewise, POSTGRES [17] implements an exhaustive search optimization algorithm. In contrast, INGRES [18] introduced an optimization technique based on query decomposition. Estimation of conditions are often supported by histogram distributions of attribute values [10]. More recently, developments in deductive and object oriented database technology showed the need for more cost-effective optimization techniques [16] as the traditional techniques work well for queries with only a few relations to join. Steinbrunn *et al.* summarizes and analyzes in [16] randomized algorithms for the problem of query optimization where the overall goal is to search the solution space for the global minima moving randomly between connected solutions according to certain rules. Further, the authors describe deterministic, genetic and hybrid algorithms as techniques for the problem of cost-effective query optimization. PostgreSQL is and example of an open source databases system experimenting with genetic algorithms for query optimization.[1]

Related to the Semantic Web, Pérez *et al.* analyze in [11] the semantics and complexity of SPARQL. Harth *et al.* [7] investigate the usage of optimized index structures for RDF. The authors argue that common RDF infrastructures do not support specialized RDF index structures. The index proposed by the authors supports partial keys and allows selectivity computation for single triple patterns. Hartig *et al.* [8] present a SPARQL query graph model (SQGM) which supports all phases of query processing, especially query optimization. The authors refer to a discussion on the Jena mailing list which showed that a simple rearrangement of a SPARQL query leads to an improvement of factor 220.[2]

The concepts and ideas presented in this paper are inspired by and rely on some previous, unpublished, own work.

[1] http://www.postgresql.org/docs/8.2/static/geqo.html
[2] http://tech.groups.yahoo.com/group/jena-dev/message/21436

Listing 2: BGP of the LUBM Query 2

```
1 ?X rdf:type ub:GraduateStudent .
2 ?Y rdf:type ub:University .
3 ?Z rdf:type ub:Department .
4 ?X ub:memberOf ?Z .
5 ?Z ub:subOrganizationOf ?Y .
6 ?X ub:undergraduateDegreeFrom ?Y .
```

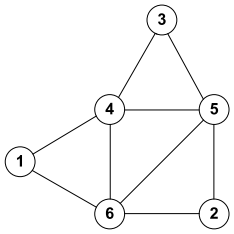

Figure 1: Undirected connected graph $g_1 \in \mathcal{G}$

In [4, 3] the authors describe the fundamental techniques which are further extended in this paper.

3. THE OPTIMIZER

In this section, we first discuss the preliminaries and formalize the theory underlying the optimizer and the heuristics. Second, we present the architecture of the optimizer and we describe its components. Finally, we discuss the heuristics used for selectivity estimation of graph patterns.

3.1 Preliminaries

For a better understanding, we develop the theory by means of the example BGP displayed in Listing 2 (i.e. the `WHERE` clause of the Lehigh University Benchmark (LUBM) [6] Query 2).

Given a BGP, B, we define B to be a graph G as a set $\mathcal{G}$ of undirected connected graphs. The elements $g \in \mathcal{G}$ are the components of G. For each pair $(g_i, g_j) \in \mathcal{G}$, g_i and g_j are disconnected. Note that the elements $g \in \mathcal{G}$ have different semantics than RDF graphs, i.e. the nodes of $g \in \mathcal{G}$ are triple patterns.

A graph $g \in \mathcal{G}$ is represented as an ordered pair $g := (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is an unordered set of distinct triple patterns (i.e. the nodes of g) and $\mathcal{E}$ is an unordered set of distinct triple pattern *pairs* (i.e. the edges of g).

A triple pattern pair shares at least one bound or unbound component. The subject, predicate, and object are the components of a triple (pattern) [9]. In this paper, we refer to triple pattern pairs as *joined* triple patterns. Hence, two triple patterns with a shared variable are joined as well as two triple pattern with, for instance, the same subject URI.

In Figure 1, we display the undirected connected graph $g_1 \in \mathcal{G}$ for the BGP in Listing 2. As the BGP triple patterns in Listing 2 are (transitively) joined, the graph G has only one component, thus $\mathcal{G}$ contains only the connected graph g_1. Note, that the numbers used for the nodes of g_1 in Figure 1 correspond to the numbers of the triple patterns of the BGP in Listing 2. Further, note that, for the sake of simplicity, in Figure 1 (and generally in our working example) we consider only joins defined by unbound components (i.e. variables). This is a specialization of the more general case described in this section as, in our example, we do not consider bound components shared by two triple patterns, e.g. the bound `rdf:type` predicate.

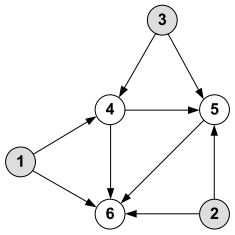

Figure 2: DAG d_g for Listing 2 with highlighted nodes with only outgoing directed edges

For a BGP B the execution order of pairwise disconnected graphs $(g_i, g_j) \in \mathcal{G}$ does not affect query performance as the overall result set corresponds to the Cartesian product of the result sets for g_i and g_j. Therefore, we can reduce the optimization problem for B to the optimization of each $g \in \mathcal{G}$. In the following, we focus on the optimization of connected graphs $g \in \mathcal{G}$.

Definition 1. The size N of $g \in \mathcal{G}$ is the number of nodes of g, i.e. the number of triple patterns in g.

Definition 2. An execution plan p_g for $g \in \mathcal{G}$ is a well defined order for the nodes of g.

Definition 3. The set $\mathcal{P}_g$ is the execution plan space of $g \in \mathcal{G}$. An execution plan $p_g \in \mathcal{P}_g$ is an element of the space. The size of $\mathcal{P}_g$ is the total number of execution plans for $g \in \mathcal{G}$.

Given the size N of $g \in \mathcal{G}$, the size of $\mathcal{P}_g$ is $N!$ (on single processor machines).[3] Therefore, the expanded execution plan space $\mathcal{P}_g$ is potentially huge even for a simple BGP with only a few triple patterns.

An execution plan p_B for a BGP B is an unordered set, $\mathcal{Q}$, whose elements are execution plans p_g. The size of $\mathcal{Q}$ equals the size of $\mathcal{G}$, i.e. every connected graph $g \in \mathcal{G}$ has an associated execution plan $p_g \in \mathcal{P}_g$ which is an element of $\mathcal{Q}$.

An execution plan $p_g \in \mathcal{P}_g$ can be represented as a directed acyclic graph (DAG). We define $\mathcal{D}_g$ as the set of directed acyclic graphs for the execution plans in $\mathcal{P}_g$. Each DAG $d_g \in \mathcal{D}_g$ represents one or more execution plans p_g of an undirected connected graph $g \in \mathcal{G}$. In Figure 2 we show the DAG corresponding to the execution plan p_g which executes the BGP of Listing 2 top-down, i.e. the triple patterns are evaluated in the same order as they are listed in Listing 2. For any two nodes $(i, j) \in d_g$, there is a directed path between i and j, if (1) the triple patterns corresponding to i and j are (transitively) joined and (2) i is executed first in the execution plan. There is a clear relationship between the set $\mathcal{P}_g$ of execution plans p_g and the set $\mathcal{D}_g$ of directed acyclic graphs d_g. More formally, we can state the following function $f : p_g \rightarrow d_g$ that is injective and not surjective. Thus, an execution plan p_g can be mapped uniquely to a DAG d_g, whereas a DAG d_g is an abstraction for one or

[3]There are $N!$ plans for N triple patterns when considered to be executed linearly. On truly parallel systems, we have the option to execute patterns in parallel and so there are at least $N!$ plans.

more execution plans p_g. For instance, the execution plan which executes the triple patterns of the BGP in Listing 2 top-down, i.e. the sequence $(1, 2, 3, 4, 5, 6)$ of triple patterns, is uniquely mapped to the DAG displayed in Figure 2. However, the DAG in Figure 2 is also an abstraction for the execution plan expressed by the sequence $(2, 3, 1, 4, 5, 6)$ of triple patterns. The size of $\mathcal{D}_g$ is, hence, generally not equal to the size of $\mathcal{P}_g$.

3.2 The Architecture

In this section, we present the architecture of the optimizer implemented in ARQ. The optimizer consists of three main components: (1) the BGP abstraction, (2) the core optimization algorithm, and (3) the extensible pool of selectivity estimation heuristics.

The selectivity estimation of graph patterns is fundamental for the optimization of basic graph patterns. According to [12], the selectivity of a condition is the fraction of tuples satisfying the condition. In our domain, the selectivity of a (joined) triple pattern is the fraction of triples matching the pattern. For a typical dice, the selectivity of even numbers is 0.5. Refer to Section 5 for a detailed discussion about RDF selectivity estimation for our purpose of BGP optimization.

BGP Abstraction. As discussed in Section 3.1, we abstract a BGP as an undirected graph B which is characterized by the connected components $g \in \mathcal{G}$, where each g is an ordered pair $g = (\mathcal{N}, \mathcal{E})$ consisting of a set $\mathcal{N}$ of triple patterns (i.e. the nodes of g) and a set $\mathcal{E}$ of triple pattern pairs (i.e. joined triple patterns/edges of g). The connected graph $g \in \mathcal{G}$ represents a subset of (transitively) joined triple patterns of B. In the following we describe the algorithm for the optimization of $g = (\mathcal{N}, \mathcal{E})$.

Based on the BGP abstraction for $g \in \mathcal{G}$, we perform a variation of the deterministic minimum selectivity approach [16] to identify the execution plan p_g which is optimal according to the algorithm and the selectivity estimations. The optimization algorithm constructs a solution in a deterministic manner applying a heuristic search. Eventually, the algorithm identifies an order for the elements of the set $\mathcal{N}$ (i.e. triple patterns). Note that this is not a total order on $\mathcal{N}$. The triple patterns in the resulting execution plan are not necessarily ranked by estimated selectivity.

Optimization Algorithm. In Algorithm 1, we provide the pseudo-code for the core optimization algorithm. The algorithm first selects the edge with minimum estimated selectivity from $g = (\mathcal{N}, \mathcal{E})$. The corresponding nodes are marked as visited and added to the final execution plan p_g ordered by estimated selectivity, i.e. the more selective node is added first to the execution plan. After selecting the first edge $e \in \mathcal{E}$, the core optimization algorithm iteratively selects the edge which satisfies the two properties (1) minimum estimated selectivity and (2) visited node. With each iteration a new node is added to the final execution plan.

The property of minimum estimated selectivity is motivated in the deterministic minimum selectivity optimization approach according to which good solutions are generally characterized by selective intermediate results [16]. The second property, i.e. visited node, ensures the iterative selection of a triple pattern, i.e. a node $n \in \mathcal{N}$, which joins with the previous partial execution plan. This is an important characteristic of good execution plans as result sets will never be the Cartesian product of two intermediate result sets. Therefore, at each stage of query processing the intermediate result sets are iteratively constrained.

Algorithm 1 Find optimized execution plan EP for $g \in \mathcal{G}$

```
N ← Nodes(g)
E ← Edges(g)
EP[size(N)]
e ← SelectEdgeMinSel(E)
EP ← OrderNodesBySel(e)
while size(EP) ≤ size(N) do
  e ← SelectEdgeMinSelVisitedNode(EP, E)
  EP ← SelectNotVisitedNode(EP, e)
end while
return EP
```

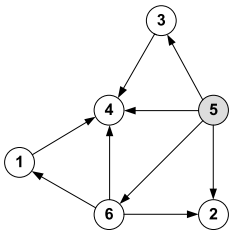

Figure 3: Optimized DAG $d_1 \in \mathcal{D}$ with highlighted node with only outgoing directed edges

The algorithm terminates when all nodes $n \in \mathcal{N}$ have been visited and the optimal execution plan p_g, i.e. a well defined order for the elements of $\mathcal{N}$, is returned as a result.

Directed acyclic graphs of execution plans which satisfy the second property, i.e. visited node, of the edge selection process for BGP abstractions described above, feature a common characteristic: there is only one node that has only outgoing directed edges, i.e. the node which is executed first in the execution plan. Nodes with only outgoing directed edges do not join with the previous partial execution plan and, hence, result in a Cartesian product of two intermediate result sets. For instance, the execution plan which executes the triple patterns of Listing 2 top-down, abstracted as DAG in Figure 2, creates two Cartesian products for the intermediate result sets of the first three triple patterns (highlighted in Figure 2 by the three nodes labeled 1, 2, and 3 which are nodes with only outgoing directed edges). In contrast, the optimized execution plan, abstracted as DAG in Figure 3, does never create Cartesian products of intermediate result sets. This is highlighted by the DAG in Figure 3 with one node with only outgoing directed edges (i.e. node 5). This node represents the first triple pattern in the optimized execution plan for the BGP in Listing 2.

Selectivity Estimation Heuristics. In order to decide the selection of edges during the optimization process, the core optimization algorithm requires figures about the selectivity [12] of graph patterns. The extensible pool of selectivity estimation heuristics is the component intended to provide the required selectivity figures to the core optimizer.

Heuristics are used to weight the nodes and edges of a BGP abstraction. Given a weighted connected graph $g \in \mathcal{G}$ the core optimization algorithm is able to proceed with the iterative selection of nodes based on the deterministic minimum selectivity optimization approach described above.

3.3 Heuristics

In this section, we present the heuristics implemented and used by the optimizer for the selectivity estimation of graph patterns. We categorize the heuristics according to whether or not they require *pre-computed* statistics about the RDF data.

Heuristics Without Pre-computed Statistics. The simplest heuristic, ARQ/VC, is called *variable counting*. For this heuristics, the selectivity of a triple pattern is computed according to the type and number of unbound components and is characterized by the ranking $sel(S) < sel(O) < sel(P)$, i.e. subjects are more selective than objects and objects more selective than predicates. Whether subjects are more selective than objects (or predicates) effectively depends on the RDF data. In typical RDF datasets, there are more triples matching a predicate than a subject or an object. The distinction between subject and object is more difficult. The ranking we have chosen is clearly a pragmatic choice. The selectivity of a *joined* triple pattern is computed according to the type and number of joins (either bound or unbound). Bound joins are considered more selective than unbound joins and subject-subject joins are more selective than subject-object or object-object joins. Unusual joins (e.g. subject-predicate) are considered the most selective. Note that the specific choice on how to estimate the selectivity of (joined) triple patterns without any statistics is certainly arguable.

The *variable counting predicates* heuristic, ARQ/VCP, is very close to ARQ/VC. In fact, the only difference is a default selectivity of 1.0 for triple patterns joined by bound predicates. For certain queries, estimating patterns joined by bound predicates as selective can be a bad choice. For instance, consider the two patterns `[?x rdf:type C]` and `[?y rdf:type D]`. Typically, both patterns return a considerable result set and it is generally a bad choice to estimate the pattern as selective. Refer to our evaluation in Section 6 for a practical example where this heuristic has a high impact on query performance (LUBM Query 2).

The *graph statistics handler*, ARQ/GSH, is the heuristic with the most accurate estimation for the selectivity of triple patterns. However, the selectivity estimation of joined triple patterns is not supported (instead we use ARQ/VC). The high accuracy of this heuristic is enabled by the underlying Jena [5] in-memory graph which enables for certain graph implementations to look-up for exact size information of any triple pattern component. We use such information to compute selectivities. The Jena graph implementation for non-inference in-memory models supports the look-up for the number of triples matching either a subject, a predicate or an object of a triple pattern. Note that, combinations (e.g. subject-predicate) are not supported by the graph implementation. Moreover, Jena graph implementations for inference models currently do not support graph statistics. In absence of a graph statistics handler for the graph the optimizer chooses the default heuristic, ARQ/VC.

Heuristics With Pre-computed Statistics. The probabilistic framework (PF) is a standalone framework for the selectivity estimation of RDF graph patterns (Section 5). It implements selectivity estimation techniques based on statistics about RDF data (Section 4). A wrapper around the PF implements the ARQ/PF heuristic for the optimizer enabling the selectivity estimation of triple patterns and joined triple patterns. Although, the heuristic requires statistics about the RDF data which have to be previously computed, heuristics based on the PF are, in average, the most accurate. Whereas ARQ/GSH only supports non-inference graph models, we can build the statistics required for the PF by computing the summary statistics of an inference model. Moreover, heuristics based on the PF are the most accurate for the selectivity estimation of joined triple patterns as the PF creates customized statistics about joined RDF graph patterns.

Based on the probabilistic framework, we implement another heuristic, called ARQ/PFJ. There are two differences to ARQ/PF. First, the estimated selectivity for joined triple patterns is a function of the estimated join selectivity *and* the selectivity of the more selective triple pattern involved in the join. This is slightly different to ARQ/PF where the selectivity for joined triple patterns is a function *only* of the estimated join selectivity. Second, ARQ/PFJ does not consider bound predicates, similar to ARQ/VCP.

The last presented heuristic is called ARQ/PFN and is a variation of ARQ/PF, hence, it is also based on the PF. The primary goal of the PF is to estimate as accurate as possible the selectivity of graph patterns (with one or two triple patterns). Since the estimated selectivity might be lower than the minimum meaningful selectivity for some RDF data (i.e. the selectivity of matching one triple) the probabilistic framework limits the estimated selectivity to this minimum. As limiting the lower bound for the estimated selectivity has a positive effect for the accuracy of the framework, it might have a negative effect for our purpose of optimization. In fact, the estimation error does not affect the performance of the optimizer as long as the order of triple patterns is correct. Hence, although the estimation might be off the boundaries, the performance of the optimizer in ordering triple patterns might be higher. Based on this observation, ARQ/PFN does not limit the lower bound of selectivity estimation (whereas ARQ/PF does). LUBM Query 12 in our evaluation in Section 6 is an example where limiting forces the optimizer to select a wrong order for the triple patterns.

4. SUMMARY STATISTICS FOR RDF

Maintaining summaries for the statistics about the data stored in databases has a long tradition in database technology. Such meta information on the distribution of data is used as an efficient way to estimate cardinalities during query processing for a number of tasks (e.g. static query optimization, result set size estimation). Histograms are typically used to represent the distribution of data [13]. In this section, we describe in detail the statistics required by the probabilistic framework (PF) to estimate the selectivity of (joined) triple patterns.

The need for customized summary statistics of RDF data for the purpose of BGP optimization is motivated by at least the following two arguments. First, our architecture discussed in Section 3 requires selectivity information to weight the nodes $n \in \mathcal{N}$ (i.e. triple patterns) and edges $e \in \mathcal{E}$ (i.e. joined triple patterns) for every connected component $g \in \mathcal{G}$ of the graph G, the abstraction of the BGP B. Second, RDF graph patterns are typically characterized by a large number of joins. The *attribute value independence assumption* discussed in database literature [13] is perhaps the most difficult challenge also for the cardinality estimation of

RDF graph patterns. Therefore, tailored summaries for the statistics of RDF data are required to accurately estimate the selectivity of graph patterns.

We have chosen a pragmatic solution which is shown to be a reasonable compromise between statistical expressivity, achieved optimizations, and the size of the summary. In the following, we first discuss the required statistics for the selectivity estimation of triple patterns. Afterwards, we focus on the statistics for *joined* triple patterns.

4.1 Triple Pattern Statistics

Triple pattern components may be bound (i.e. concrete) or unbound (i.e. variable). Thus, we have to consider two dimensions: the first dimension includes the subject, the predicate, and the object of a triple pattern, whereas the second dimension differentiates between bound and unbound components.

The case of unbound components is trivial as they do not affect the required statistics. However, we need to differentiate the case of bound components as they require different statistics. Note that in the following we refer to the number of triples matching a pattern as the *size* of the pattern. As discussed in Section 5, the size is strongly related to the selectivity. Hence, given an approximation for the size, it will be straightforward to compute the selectivity.

The size of a *bound subject* is approximated as the average number of triples matching a subject, i.e. the average number of triples for an RDF resource. Thus, the total number of triples and the total number of distinct subjects are the statistics required to estimate the size of a bound subject. The size of a *bound predicate* is the number of triples matching the predicate. We compute the exact statistics for each distinct predicate contained in the summarized RDF data, i.e. the exact number of triples matching a predicate. Finally, the size of a *bound object* is approximated by means of classical equal-width histograms [12]. For each distinct predicate we compute a histogram to represent the corresponding object-value distribution.[4]

4.2 Joined Triple Pattern Statistics

Schema ontologies provide useful information about the relations of classes and properties of vocabularies used in RDF data.

For our purpose, we especially look at `rdf:Property` instances which are related together because of a matching `rdfs:Class` instance for their domain or range. For instance, a property p_1 with domain C and a property p_2 with domain C are related because they both define the same class C as domain. We take into account not only matching domains but also domain and range or only ranges.

Given two related properties p_1, p_2 and their join relation (relation type), i.e. whether they define the same class C for their domains, domain/range, range/domain or both ranges, we compute the size of the corresponding BGP as the result set size of a SPARQL query. For instance, for p_1 and p_2 with a relation type SS (i.e. the two properties define the same class C for their domains), the corresponding BGP might be described by the two triple patterns `[?x` p_1 `?y]` and `[?x` p_2 `?z]`. The returned size is used as an upper bound for the size of any BGP involving the two properties p_1 and p_2 with join relationship SS.

In real world queries, joined triple patterns do not always have unbound subjects and objects. For instance, the BGP B with the two triple patterns `[?x` p_1 C`]` and `[?x` p_2 `?y]` (where C is a bound object) is potentially more selective than the BGP B' with the patterns `[?x` p_1 `?y]` and `[?x` p_2 `?z]`. Though, the summary supports accurate size information only for the BGP B'. In order to consider the object restriction for B, the size of B is a function of the upper bound size (i.e. the size of B') and the size of the restricted object (which is a look-up in the histogram of p_1 for C).

Often the schema of an ontology is not available, which makes it impossible to get the required insights on related properties. For this reason, we create a full summary of all combinations of distinct predicates defined in the summarized RDF data. Certainly, a schema would greatly reduce the summary size because related predicates are explicitly defined. However, in absence of the schema, it would not be possible to create the summary.[5]

4.3 Summary Features

Summarizing statistics for RDF data is a process which is performed separately from and prior to query execution. Thus, statistics are, generally, not computed during query evaluation. The size of the summary, i.e. the required memory space, is most influenced by the number of distinct predicates as they characterize the number of histograms and joined patterns indexed in the summary. As long as the number of distinct predicates is constant, a growing RDF dataset does not affect the size of the summary. A growing RDF dataset does, however, affect the figures representing the statistics in the summary. The size of the summary is typically of multiple orders of magnitude smaller than the size of the RDF dataset. We represent the summary in RDF and typically serialize it as RDF/XML [2].

Gathering the statistics on joined triple patterns (Section 4.2) is the most expensive task in the process of summary computation. The number of entries in the summary is a quadratic function of the number of distinct predicates, more precisely $f(n) = 4n^2$, where n is the number of distinct predicates. For instance, for an ontology with 14 distinct predicates, the summary for joined triple patterns is of size 784.

In real world settings, ontologies often have more than just 10 or 20 distinct predicates. For this reason, we allow specific configurations (i.e. manual intervention) to potentially improve the performance. The summary can be computed either with *full support* for both triple pattern and joined triple pattern selectivity estimation or with *partial support* for selectivity estimation of triple patterns only. Building the statistics to support triple pattern selectivity estimation is, compared to the full summary, much faster. However, it is a trade-off between time and accuracy as the accuracy of joined triple pattern selectivity estimation will be lower with a partial summary. Further, the summary supports a property exclusion list. Thus, we can specify a set of properties

[4]For practical reasons, we use the hash code of the lexical form of objects. Hence, we create histograms of integer values which allow comfortable computation of histogram class sizes as well as upper and lower bound of histograms and histogram classes. Certainly, a data type specific implementation would support selectivity estimation for ranges which is particularly interesting for the selectivity estimation of SPARQL variables constrained by the `FILTER` operator.

[5]A trivial optimization would be to consider the schema if it is available and consider all predicate combinations if the schema is unavailable.

which are ignored during the process. This is useful especially in environments where the query patterns are known. The technique may potentially significantly reduce the time required to built the summary without loss of accuracy for the specific queries. Finally, we support random sampling of RDF data. Given a set of triple statements T and a sampling percentage, the sampling technique randomly selects triple statements from T to create a new set of statements of the size according to the given sampling percentage. This new set will be used to compute the statistics. Sampling is potentially useful for large RDF datasets at the cost, however, of a loss of accuracy. Note that it is not our intention to comprehensively analyze sampling techniques for RDF data in this paper.

5. RDF SELECTIVITY ESTIMATION

In this section we describe the framework for RDF selectivity estimation. The probabilistic framework (PF) is intended to provide selectivity information for (joined) triple patterns of basic graph patterns.

The framework builds on top of the summary statistics for RDF data discussed in Section 4 and implements a simple interface that provides two methods: one for the selectivity estimation of triple patterns and one for the selectivity estimation of joined triple patterns. Heuristics based on the probabilistic framework (Section 3.3) implement wrappers around these two methods and, hence, provide selectivity information for the optimization of basic graph patterns.

The selectivity of a pattern is strongly related to its probability. Both the selectivity and the probability accept real values of the interval $[0, 1]$. Selective patterns, e.g. `[ex:s ex:p ex:o]`, have a selectivity value that approaches the lower bound of the interval, i.e. 0. Selective patterns have a low probability to match triples in some RDF dataset. On the contrary, unselective patterns, e.g. `[?s ?p ?o]`, have a selectivity value that approaches the upper bound of the interval, i.e. 1. Unselective patterns have a high probability to match triples in some RDF dataset. This definition is valid for both triple patterns and joined triple patterns. Note that this might be in contrast with the definition for selectivity in other parts of the literature, in which the selectivity is the inverse of the probability.

5.1 Triple Pattern Selectivity

The selectivity of a triple pattern is estimated by the formula $sel(t) = sel(s) \times sel(p) \times sel(o)$ where $sel(t)$ denotes the selectivity for the triple pattern t, $sel(s)$ the selectivity for the subject s, $sel(p)$ the selectivity for the predicate p, and $sel(o)$ the selectivity for the object o. The (estimated) selectivity is a real value in the interval $[0, 1]$ and corresponds to the (estimated) number of triples matching a pattern, i.e. the *size* (Section 4), normalized by the total number of triples in the RDF dataset. Note that this formulation only approximates $sel(t)$ as it implicitly assumes that $sel(s)$, $sel(p)$, and $sel(o)$ are statistically independent, which they will not be in most cases.

The selectivity of *unbound* triple pattern components, i.e. the selectivity of a variable subject, predicate, or object, is generally 1.0 as an unbound component essentially matches every triple in the dataset.

In the following, we discuss the selectivity of *bound* components. The selectivity of a bound subject is estimated by the formula $sel(s) = \frac{1}{R}$ where R denotes the total number of resources in the RDF dataset.[6] Note that the estimated selectivity for a bound subject is constant in our model.

The selectivity of a bound predicate is computed by the formula $sel(p) = \frac{T_p}{T}$ where T_p is the number of triples matching predicate p and T is the total number of triples in the RDF dataset. Note that, as discussed in Section 4, the summary provides exact figures for T_p and T. Therefore, the selectivity of p is exact and not an estimation.

The selectivity of a bound object is estimated by the formula

$$sel(o) = \begin{cases} sel(p, o_c), & \text{if p is bound;} \\ \sum_{p_i \in \mathbb{P}} sel(p_i, o_c), & \text{otherwise.} \end{cases} \quad (1)$$

where the pair (p, o_c) represents the class of the histogram for predicate p in which object o falls, and $sel(p, o_c) = \frac{h_c(p,o_c)}{T_p}$ denotes the frequency of class (p, o_c) normalized by the number of triples matching predicate p. If the predicate is unbound, the histogram of each predicate is considered for the object selectivity estimation, i.e. the selectivity of the object is the sum of $sel(p_i, o_c)$ for each predicate $p_i \in \mathbb{P}$ in the summary.

5.2 Joined Triple Pattern Selectivity

The selectivity estimation of joined triple patterns is supported by the summary described in Section 4.2. The summary provides upper bound sizes for related predicates with unbound subjects and objects, where *size* denotes the result set size of a joined triple pattern. Given the upper bound size S_P for a joined triple pattern P, the selectivity of P is estimated as

$$sel(P) = \frac{S_P}{T^2} \quad (2)$$

where T^2 denotes the square of the total number of triples in the RDF dataset. The square of the total number of triples equals the size of the Cartesian product of two triple patterns with pairwise distinct unbound components.

The selectivity computed by Equation 2 is corrected by a specific factor for joined triple patterns with *bound* subjects or objects. This factor is a function of the selectivities for the bound components (subject and object) of the triple patterns. For instance, if the pattern P is joined over the subjects by a variable and the first triple pattern has a bound object O, i.e. P contains the triple pattern `[?x` p_1 `O]` and `[?x` p_2 `?y]`, the selectivity of P is estimated as

$$sel(P) = \frac{S_P}{T^2} \times sel(p_1, o_c) \quad (3)$$

where $sel(p_1, o_c)$ is the object selectivity of the first triple pattern in P (Equation 1).

6. EVALUATION

In this section, we present the results of our evaluation based on the Lehigh University Benchmark (LUBM) [6]. The LUBM features an OWL ontology for the university domain, enables scaling of datasets to an arbitrary size, and

[6]Note that this is in accordance with the description in Section 4.1. In fact, $sel(s) = \frac{\frac{T}{R}}{T} = \frac{T}{R} \times \frac{1}{T} = \frac{1}{R}$, where T denotes the total number of triples.

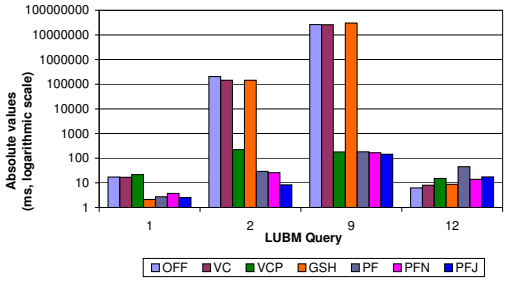

Figure 4: LUBM Query Performance

provides 14 extensional queries. Our evaluations are performed on a dataset containing the description of a single university (the LUBM University0) and the OWL-DL entailed statements. The total number of triples on which we perform the evaluations is 156,407.

First, we evaluate the query performance for the 14 LUBM queries using ARQ and the optimizer with the heuristics discussed in this paper (Section 3.3). Second, we evaluate the performance of the optimizer using summaries for the statistics of sampled LUBM datasets where samples are created by randomly selecting statements from the full dataset. Finally, in our third evaluation we explore the execution plan space of the 14 queries and identify the position that ARQ and the optimization heuristics occupy within this space. This allows us to investigate whether or not our heuristics are able to find the best execution plan for the queries. Moreover, we can compute an average distance (in execution plans) from the best performing plan for each heuristic and the 14 LUBM queries.

We align the performance of ARQ with disabled optimizer (ARQ/OFF) as a reference for the performance of the heuristics implemented by the ARQ optimizer. All evaluations are performed on a Red Hat Linux empowered AMD Opteron^{TM}dual core machine with 8GB main memory.

6.1 Query Performance

The query performance evaluation for the LUBM is summarized in Figure 4. The LUBM specifies some queries that are more interesting for the purpose of BGP optimization than others. Some queries define a BGP with a substantial number of triple patterns and joins between them, e.g. Queries 2 and 9. Others are more simple and contain just one triple pattern, e.g. Queries 6 and 14, or two triple patterns, e.g. Queries 1, 3, 5, 10, 11, and 13. Hence, it is not surprising that for certain queries no optimization is achieved at all.

Figure 4 summarizes the query performance for 4 queries of the LUBM. On one dimension we show the LUBM queries. The second dimension shows the absolute values for the query performance in milliseconds on a logarithmic scale. The figure compares the performance for ARQ with disabled optimizer (ARQ/OFF) and ARQ with the heuristics discussed in Section 3.3, i.e. the variable counting (ARQ/VC), the variable counting (no) predicates (ARQ/VCP), the graph statistics handler (ARQ/GSH), the probabilistic framework (ARQ/PF), the probabilistic framework without limit for the lower bound of selectivity estimation (ARQ/PFN) and, finally, the heuristic based on the probabilistic framework with a different selectivity estimation for joined triple patterns (ARQ/PFJ).

Query 1 is similar to our example query in Listing 1 as it defines a BGP with two triple patterns. A simple rearrangement for the two triple patterns leads to a performance improvement of one order of magnitude.

Query 2 is interesting for multiple reasons. First, the achieved optimization is of four orders of magnitude for the heuristics based on the probabilistic framework. Second, the explicit low selectivity (1.0, i.e. unselective) of triple patterns joined by bound predicates used for ARQ/VCP avoids to first execute the three triple patterns with bound `rdf:type` predicates. In fact, the three triple patterns create a huge intermediate result set which is only subsequently constrained. In contrast, ARQ/GSH does estimate the first three triple patterns of the BGP as selective and, hence, executes them first in query evaluation. This results in a poor query performance although ARQ/GSH has precise selectivity information for the components of triple patterns. Of course, we can think of a heuristic that combines ARQ/GSH and ARQ/VCP which would most likely perform better than ARQ/VCP. Note that the fact that ARQ/GSH and ARQ/VC build Cartesian products for intermediate result sets is not in contrast with the discussion in Section 3.1. If the BGP graph abstraction would not define a join for bound predicates, the algorithm would guarantee to not choose execution plans with Cartesian products as intermediate result sets. It was a design choice to define a join for every bound or unbound pair of triple pattern components to be as general as possible. In future, we might choose to ignore bound predicate joins directly in the BGP abstraction process. Finally, some words about ARQ/PF and ARQ/PFJ: the former executes the two triple patterns with empty joined result set first in the query (triple pattern 5 and 6 of LUBM Query 2). The problem is that the computation of this join is more expensive than the overall join computations for the execution plan selected by ARQ/PFJ. In fact, the first triple pattern executed by ARQ/PF matches 2414 triples, whereas the second triple pattern matches 463 triples. Although combined the result set is empty and, hence, the final result set is identified after executing the second triple pattern, the cost of executing the first join is more expensive than the total cost of three joins required to evaluate the execution plan selected by ARQ/PFJ. Note that the first triple pattern executed by ARQ/PFJ matches only 15 triples. Although, this small result set is joined another two times to get the final empty result set, it is, overall, less expensive.

The considerations for Query 9 are similar to Query 2. The achieved optimizations are of five orders of magnitude for the best performing heuristics. Again, we note that ARQ/VCP performs optimally without requiring any statistics. Clearly, it is crucial to avoid the execution of the three triple patterns with bound `rdf:type` predicates first in query evaluation. The rationale is the same as for Query 2.

Query 12 is also worth a couple of words. Whereas for all other queries ARQ/PF shows a similar performance as the other heuristics based on the probabilistic framework, for Query 12 ARQ/PF clearly selects a wrong ordering of the triple patterns. This is caused by the explicit limit of the lower bound for selectivity estimation in ARQ/PF. In fact, two triple patterns of the BGP in Query 12 are estimated with the same (lowest) selectivity by ARQ/PF. By not limiting the estimation to the lowest meaningful selectiv-

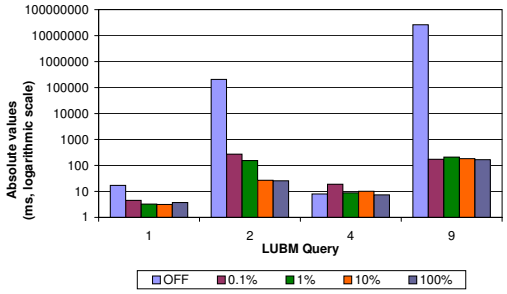

Figure 5: LUBM Sampling Performance

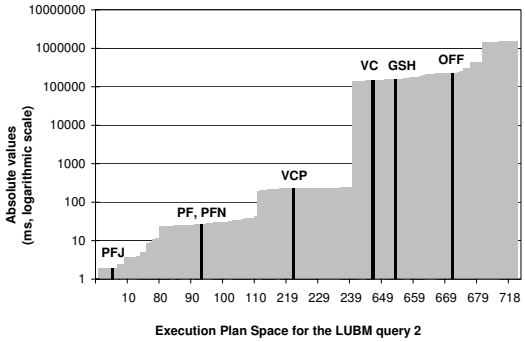

Figure 6: Plan Space for the LUBM query 2

ity (ARQ/PFN) the optimizer identifies a better ordering. Whereas the accuracy of the estimation is higher for ARQ/PF compared to ARQ/PFN, for the purpose of ordering triple patterns the limitation to the lowest meaningful selectivity is, in this case, better avoided.

6.2 Sampling Performance

The sampling performance evaluation for the LUBM is summarized in Figure 5. We use a random selection of statements from the original dataset to create three subsets containing respectively 0.1%, 1%, and 10% of the triples contained in the original dataset. We create the summary statistics for each sample and use them for query evaluation with the ARQ/PF heuristic. Generally, we can state that the summaries for the 0.1% sample are accurate enough to significantly improve query performance. Being more accurate, summaries of larger samples further improve the query performance, however, in most cases this improvement is not as significant as the improvement achieved by the 0.1% summary. Note that the LUBM is a synthetic RDF dataset which may bias the evaluation based on samples.

The main advantage of summaries for samples is the time required to create them. The time required to create the summary for the 0.1% sample is approximately 13 seconds while the time required to create the summary for the entire dataset is approximately 1,764 seconds (∼30 minutes).

Figure 5 summarizes the query performance for a subset of interesting LUBM queries. For Query 1, the summary of the 0.1% sample is clearly enough accurate to correctly reorder the simple BGP consisting of two triple patterns. Query 2 is an example where more accurate information is required to capture the peculiarity of the BGP, i.e. that the two last triple patterns together have an empty intermediate result set. A summary of 10% of the original dataset is accurate enough to optimize the query. Query 4 is an example where sampling worsens the performance compared to the original BGP, for the 0.1% sample. As we select the statements randomly, the resulting inaccurate statistics entice the optimizer to select a bad execution plan for the BGP. Finally, Query 9 is similar to Query 2 with the difference that the summary of the 0.1% sample is already enough accurate to optimize the query.

Query	Min	Max	ARQ/PFJ
2	1.88	1,532,992.17	1.95
4	1.59	218.59	1.61
7	1.36	98,474.56	1.36
8	115.56	30,382.56	116.09
12	2.74	61.84	10.66

Table 1: The best, the worst, and the ARQ/PFJ execution plan performance for the LUBM queries with at least three triple patterns (in milliseconds)

6.3 Exploring the Execution Plan Space

In this section, we evaluate the query performance for each execution plan in the plan space of each of the 14 LUBM queries. The evaluation allows us to identify the position of the execution plans selected by the different heuristics in the plan space of each query. Moreover, it enables to identify the best and worst execution plan, and to compute the performance interval between the two boundaries. Further, we can calculate the average distance from the best performing execution plan for each heuristic.

Following the discussion in Section 3.1, the BGP of the LUBM Query 2 is abstracted as a set $\mathcal{G} = \{g_1\}$ with a single connected graph g_1. The size of g_1 is the number of triple patterns, i.e. 6. The size of the execution plan space is, therefore, $6! = 720$. Figure 6 shows the query execution performance for a subset of the 720 queries (execution plans) in the execution plan space of the LUBM Query 2. We highlight the execution plans which reflect the plans selected by ARQ and the optimization heuristics.

In Table 1, we list the query performance of the best and worst execution plan and the performance of ARQ/PFJ for the LUBM queries with at least three triple patterns. The table shows that our best performing heuristic is very close to the best performing execution plan in the space of the listed LUBM queries. Note that the plan space evaluation for the LUBM Query 9 had to be terminated after over two weeks and could not be completed.

Finally, Figure 7 shows the *distance* of ARQ with disabled optimizer (ARQ/OFF) and each optimization heuristic to the best performing execution plan normalized by the size of the plan space, averaged over the 14 LUBM queries. While ARQ with disabled optimizer (ARQ/OFF) has a normalized average distance of 0.68 from the best plan, the best performing heuristic (ARQ/PFJ) has a normalized average distance of 0.023 from the best execution plan and is, hence, very close to the best performing execution plans.

The evaluation shows that although the best performing heuristic is not always able to find the best performing execution plan, it is, however, able to identify an execution plan that, essentially, performs in average equally to the optimal execution plan.

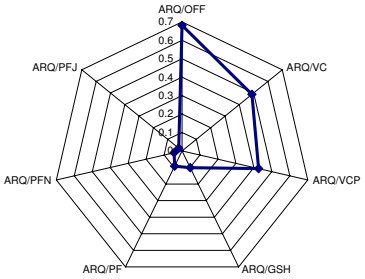

Figure 7: The normalized average distance from the best execution plan for each heuristic

7. LIMITATIONS AND FUTURE WORK

As we argued in Section 1, scaling is one of the major limitations of the presented work. Clearly, main memory graph implementations do not scale. However, we believe that the optimization of native SPARQL query engines is, nevertheless, an important issue for an efficient query evaluation on the Semantic Web.

Basic graph patterns are fundamental to SPARQL as they define the access to the RDF graph. However, they are not the only part of the SPARQL syntax interesting for static optimizations. SPARQL includes a number of operators which are modifiers for the result sets of basic graph patterns, e.g. `OPTIONAL`, `UNION`, and `FILTER`. For instance, filtered variables may be rewritten in basic graph patterns. By means of typed histograms for the distribution of object values, variables filtered by inequality operators could be considered during (joined) triple pattern selectivity estimation. A filtered variable defining the age of a person to be ≤ 10 could, for example, influence the selectivity of the (joined) triple pattern, which introduces the filtered variable. As our equal-width histograms are untyped, the optimizer is currently not able to estimate the selectivity of value ranges.

One shortcoming of the evaluation is that the underlying data is artificial. We, therefore, ran two queries which are similar to the ones used in [3] against the SwetoDBLP dataset.[7] Preliminary findings indicate that the relative performance of the different heuristics is analogous to the ones observed in the LUBM dataset.

8. CONCLUSIONS

The paper summarizes the research we have been doing on static *Basic Graph Pattern* (BGP) optimization based on selectivity estimation for *main memory* graph implementations of RDF data.

We formalized the problem of BGP optimization (Section 3.1) and we presented the architecture for the optimizer (Section 3.2) that has been implemented for ARQ. Further, we discussed a number of heuristics (Section 3.3) for the selectivity estimation of *joined* triple patterns. The heuristics range from simple variable counting techniques to more sophisticated selectivity estimations based on the probabilistic framework (Section 5) that builds on top of tailored summary statistics for RDF data (Section 4).

As the evaluation clearly showed, the characteristics of the heuristics greatly influence the selected ordering of the triple patterns of a BGP and, hence, the query execution performance. In our experience, we found the following properties of heuristics to be important for the problem of BGP optimization. First, the optimizer should avoid Cartesian products as intermediate result sets. Second, the selectivity should not be limited in lower bound estimation. Third, the selectivity of joined triple patterns should be a function of the estimated selectivity of the join (i.e. the size of the result set) *and* the selectivity of the more selective triple pattern involved in the join. Finally, as we noticed multiple times, bound predicates should not be considered as joins.

[7] http://lsdis.cs.uga.edu/projects/semdis/swetodblp/

9. REFERENCES

[1] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference. Technical report, W3C, 2004.

[2] D. Beckett and B. McBride. RDF/XML Syntax Specification. Technical report, W3C, 2004.

[3] A. Bernstein, C. Kiefer, and M. Stocker. OptARQ: A SPARQL Optimization Approach based on Triple Pattern Selectivity Estimation. Technical Report ifi-2007.03, University of Zurich, Department of Informatics, Winterthurerstrasse 190, 8057 Zurich, Switzerland, March 2007.

[4] A. Bernstein, M. Stocker, and C. Kiefer. SPARQL Query Optimization Using Selectivity Estimation. In *Poster Proceedings of the 6th International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science. Springer, 2007.

[5] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. Technical report, HP Laboratories, 2003.

[6] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2–3):158–182, 2005.

[7] A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *Proc. of the 3rd Latin American Web Congress*, page 71, 2005.

[8] O. Hartig and R. Heese. The SPARQL Query Graph Model for Query Optimization. In *Proc. of the 4th European Semantic Web Conf.*, 2007.

[9] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. Technical report, W3C, 2004.

[10] B. J. Oommen and L. Rueda. The Efficiency of Modern-Day Histogram-Like Techniques for Query Optimization. *The Computer Journal*, 45(2):494–510, 2002.

[11] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *Proc. of the 5th Int. Semantic Web Conf.*, pages 30–43, 2006.

[12] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 256–276, 1984.

[13] V. Poosala and Y. E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *The VLDB Journal*, pages 486–495, 1997.

[14] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2008.

[15] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, 1979.

[16] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal: Very Large Data Bases*, 6(3):191–208, 1997.

[17] M. Stonebraker and L. A. Rowe. The Design of POSTGRES. In *SIGMOD Conference*, pages 340–355, 1986.

[18] E. Wong and K. Youssefi. Decomposition: A Strategy for Query Processing. *ACM Trans. Database Syst.*, 1(3):223–241, 1976.