How to bring some MAGIC to SPARQL

Work in progress Paper

Christina Ehrlinger
University Passau
Germany
Christina.Ehrlinger@uni-passau.de

ABSTRACT

For SPARQL, the query language to retrieve information from an RDF graph, the optimization of the order of the containing triples is a challenging topical issue. In this paper, we show an approach which passes the information about the already found binding for a variable to all other occurrences of this variable in combination with a cost model in order to minimize the execution time. Different to other approaches our approach as described in this paper can be applied to a SPARQL query, which not only consists of basic graph pattern, but also contains group graph pattern like FILTER or OPTIONAL. Our experiments show the applicability of our approach and function as preliminary proof of concept.

1. INTRODUCTION

The structure of the web changes from linked documents, which contain the information, to directly link the information using the keyword Linked Open Data. This direct linking of information generates a huge number of quite big graphs. In order to handle these graphs, the Resource Description Framework RDF [1] is one of the used standard models for data interchange on the Web. It uses Uniform Resource Identifiers (URIs) to refer to resources in order to connect them. The resulting structure is a directed, labeled graph, where the edges represent the connection between the resources. These RDF graphs are generated very easily due to the absence of a schema. There exist specialized databases for storing these RDF graphs, so-called triple stores [2].

The SPARQL query language [8] is used to access the information stored in these RDF graphs. These SPARQL queries must be performant in order to find the desired information in these huge graphs. For optimizing the execution time of a SPARQL query we can consider two different approaches. First, we can design a triple store, which optimizes the storing or the usage of indices. There are also approaches which focus on this like the RDF-3X triple store [7]. Second, we can rewrite the SPARQL query in order to optimize the order of the triples.

The same kind of optimizations took place for SQL. There are highly performant database systems which also considers the storage of the relations as well as techniques to optimize the order of the joins, which have to be done in order to execute SQL queries with more than one relation.

The aim of this approach is not an optimization, which perfectly fits into a self-designed triple store. It is a generic way how to reorder SPARQL queries in order to use their full potential according to their execution time.

The rest of the paper is organized as follows. Section 2 describes the problem statement for the reordering of the triples and states a motivational example. Section 3 presents our approach for using the sideways information passing first for a query containing a basic graph pattern and furthermore the adaption of the approach in order to use other components besides to a basic graph pattern, for example FILTER. Section 4 shows the experiments which were executed so far. Section 5 discusses related work.

2. PROBLEM STATEMENT

In order to present our approach for reordering a SPARQL query, we first have a closer look at a simple SPARQL query and discuss the impact of different orderings. Considering the SPARQL query in Listing 1. This query can be executed against a dataset generated using the Lehigh University Benchmark (LUBM) data generator [4]. It consists of 5 triples, whereas triple `t5` also contains a literal.

```
SELECT *
WHERE {
    ?p :teacherOf ?c.               #t1
    ?s :takesCourse ?c.             #t2
    ?s :advisor ?p.                 #t3
    ?s :memberOf ?d.                #t4
    ?d :name 'Department0'.         #t5
}
```

Listing 1: Example for a SPARQL query

[1]https://www.w3.org/RDF/
[2]https://www.w3.org/wiki/LargeTripleStores

31st GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 11.06.2019 - 14.06.2019, Saarburg, Germany.
Copyright is held by the author/owner(s).

CEUR-WS.org/Vol-2367/paper_9.pdf

While executing the query as shown in Listing 1 using the common triple store *GraphDB*[3], the query has an average execution time of 4606 ms while executing ten times. If we slightly change the order of the triples, we get the query shown in Listing 2, whereas this query has an average execution time of 12283 ms (executed ten times). Even for this simple query executed against a quite small dataset of approximately 130 000 triples we see the huge impact of reordering the triples of a SPARQL query.

```
SELECT *
WHERE {
    ?s :takesCourse ?c.          #t2
    ?p :teacherOf ?c.            #t1
    ?s :memberOf ?d.             #t4
    ?s :advisor ?p.              #t3
    ?d :name 'Department0'.      #t5
}
```

Listing 2: Reordered SPARQL query from Listing 1

We also find examples of reordering which lead to a high improvement regarding the execution time. In Listing 3 we again have the same SPARQL query apart from the ordering of the triples. This query has an average execution time of 325 ms.

```
SELECT *
WHERE {
    ?d :name 'Department0'.      #t5
    ?s :memberOf ?d.             #t4
    ?s :advisor ?p.              #t3
    ?s :takesCourse ?c.          #t2
    ?p :teacherOf ?c.            #t1
}
```

Listing 3: Reordered SPARQL query from Listing 1

The mechanism of reordering to get a more efficient query is not a new idea. This was done in Datalog [5] while applying the Magic Set Transformation [1] to a Datalog program in order to sort the subgoals of a rule and the same idea was also used while applying the join order optimization to a SQL query in a relational database system [11].

Especially for Datalog and SPARQL, the ordering of subgoals or respectively triples has a huge impact on the variables. The term *Sideways Information Passing*, short SIP, was used for this in Datalog and it describes how the bindings of variables are passed from one subgoal to another subgoal. Depending on the order of the subgoals it results in a variety of different SIP possibilities, called SIP strategies [10]. For our approach, we adopt this term for SPARQL. In the context of SPARQL, it describes the way the binding of the variables is passed between the triples.

Consider the query from Listing 3 again. We start with the triple `?d :name 'Department0'`. This generates bindings for the variable `?d`, which are passed to the second triple `?s :memberOf ?d`. Again this generates bindings for the variable `?s`, which is passed to the triples `?s :advisor ?p` and `?s :takesCourse ?c`.

[3]https://www.ontotext.com/products/graphdb/

In the following chapter, we present an approach in order to use the sideways information passing for SPARQL queries.

3. USING SIDEWAYS INFORMATION PASSING FOR SPARQL QUERIES

In order to optimize a SPARQL query regarding the SIP, we can represent the query as a graph to visualize the possible ways to pass the bindings. In Figure 1 we see the resulting graph for the SPARQL query in Listing 1.

Every node represents a triple pattern of the query. Two nodes are connected with an undirected edge if they share a common variable. Conceptually, the representation allows us to determine the best order of the triples by transforming the undirected graph in a directed graph using the approach described in this paper, whereas the directed edge between node $t1$ and node $t2$ denotes that $t1$ is in the ranking order before $t2$.

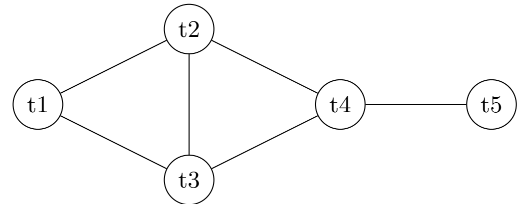

Figure 1: Query graph for query in Listing 1

Another way to visually represent the query is shown in Figure 2. Here we represent every subject and object position as a node, no matter if this is a variable, an IRI or a blank node. For the triple pattern

$$?s\ takesCourse\ ?c$$

we get two nodes representing the variables $?s$ and respectively $?c$ and they are connected with a directed edge, which represents the predicate *takesCourse* between these two variables. The edge is directed from the node representing the subject of the triple to the node representing the object of the triple. For the triple $?s$ *takesCourse* $?c$, the edge is directed from the node representing the variable $?s$ to the node representing the variable $?c$.

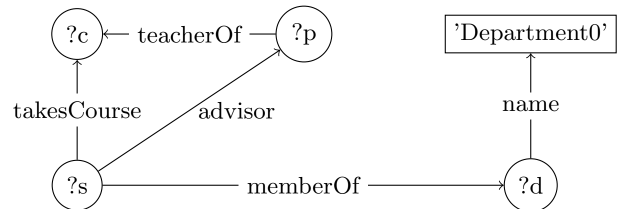

Figure 2: SPARQL graph for query in Listing 1

This kind of representation describes the pattern we are looking for while executing the query against a dataset.

The approach described in this paper focuses on how to rewrite the query with a more efficient ordering of the triples. This is done based on the query graph in order to optimize the sideways information passing.

It consists of the following four steps:

1. Generate query graph
2. Transform undirected graph into directed graph
3. Determine order of the nodes according to the direction of the edges
4. Generate optimized SPARQL query

According to the different elements of a SPARQL query, we have a closer look at the rearrangement of a query only consisting of a basic graph pattern and in a second step, we also consider all possible group graph patterns of a SPARQL query.

3.1 Basic Graph Pattern Query

Basic graph patterns (BGP) are sets of triple patterns [8]. The most simple SPARQL query consists of exactly one triple pattern:

```
?s ?p ?o
```

One example of a query which consists only of a basic graph pattern but more than one triple pattern is shown in Listing 1. As described before, in the first step we generate the corresponding query graph for the SPARQL query from Listing 1 as seen in Figure 1. During the second step, we convert the undirected query graph into a directed graph. In the following, we describe how this is achieved. Similar to Datalog, while performing the Sideways Information Passing as a preliminary step before the Magic Set Transformation [1], we try to find Literals or IRIs in the query to get a starting point for generating bindings for the variables. In general, we are looking for the node with the highest number of bound positions (subject, predicate, object). In our example, the only literal is the name of the department, so the triple, which contains this literal, will be the first triple according to the ranking order.

As explained before, this generates a binding for the variable `?d`. This binding is passed to all triples, which also contain the variable `?d`, no matter, if it occurs in the subject or object position. In the query graph, this passing is visualized by adding an outgoing direction to all undirected edges of the current node, which corresponds to the current triple. The search for a node, where the corresponding triple has the highest number of bound positions and the subsequent adding of directions to edges are repeated until all edges are directed.

This approach works fine for SPARQL queries, where the corresponding query graph is basically a path. In our example query, we can follow the path until node $t4$. Now more than one node is affected by adding the direction of an edge and we have to decide which of the two nodes $t2$ and $t3$ should be processed next. In order to solve this, we have introduced a cost-based model to pick the best next node out of all possible nodes.

The chosen cost model takes into account the average occurrence of every property but in addition, also considers the position of the common variable. For every property p we determine two different numbers, so-called *outgoing* and *incoming* value. The naming of these values origins in the SPARQL graph representation. The outgoing value describes the average amount of outgoing edges of this property

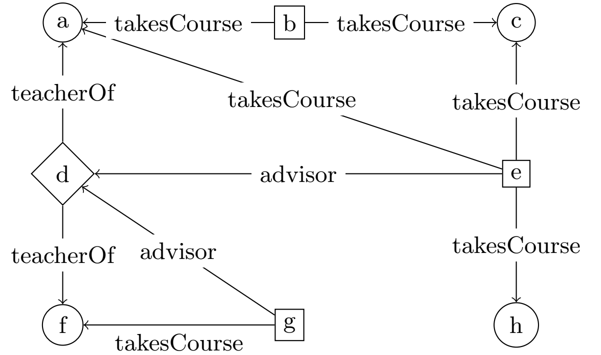

Figure 3: small sample instance graph as generated by LUBM

for a node if this node has at least one outgoing edge of this property. In general, the outgoing value for property `p` determines the average number of awaiting bindings for the object position while the subject position already has a binding, whereas the incoming value determines the average number of awaiting bindings for the subject position while the object position already has a binding.

We also have to consider the cost model in order to choose the first node, if the query contains only triples of the form `?s p ?o`, where only the predicate position `p` is bound and not a single literal or IRI is part of the query.

In Figure 3 we see a small instance graph as an example of the generated RDF graphs by LUBM. The shape of the node represents the type of the node. A rectangle represents a student, a triangle represents a professor and a circle represents a course. The outgoing and incoming values resulting from the small example can be seen in Table 1.

property	*outgoing*	*incoming*
advisor	1	2
takesCourse	2	1.5
teacherOf	2	1

Table 1: Outgoing and incoming values calculated based on the instance graph in Figure 3

For a better understanding, we have a closer look at the calculations for the outgoing and incoming values. Consider the property `takesCourse`. We sum up all edges with the label `takesCourse`. Node `e` has three edges, the node `b` has two edges and the node `g` has one edge with the label `takesCourse`. Overall we have six edges with this label. To determine the outgoing value we divide this sum by the number of nodes, which have an outgoing edge with this label (in this example three nodes). Overall we get an outgoing value of 2 for the property `takesCourse`. Similar calculations are done for the incoming value for the property `takesCourse`. Now we consider all nodes, which have an incoming edge with this label. Node `a` has two edges as well as node `c`. Both nodes `f` and `h` have one edge. Again the number of edges is divided by the number of nodes in order to calculate the average amount. Overall we get an incoming value of 1.5 for the property `takesCourse`.

So we have two different values for every property. Due to this distinction, we take into account the sideways informa-

tion passing to the subject or rather the object position.

Using this cost model we can now determine which node to choose in our example from Figure 1. After handling the nodes `t5` and `t4` we have to choose between the nodes `t2` and `t3`. In both nodes the corresponding triple contains the already bound variable in the subject position, so we compare the outgoing values of both predicates as seen in Table 1 and choose the smaller one in order to minimize the intermitted results. Because `advisor` has an outgoing value of 1 while `takesCourse` has an outgoing value of 2, we choose node `t3` to be the subsequent processed node.

Based on the number of bound positions and if equivocal based on the presented cost model we are able to transform the undirected query graph into a directed graph. In the third step, we can use topological sorting for extracting the resulting order of the nodes from the directed graph. In the last step, we map the node to the corresponding triple in order to get the optimized SPARQL query.

3.2 Handling FILTERs

Until now we have considered SPARQL queries which consist of a basic graph pattern or in other words of a set of triple patterns. On the one hand, these triple patterns generate bindings for variables. On the other hand, they are also consuming the binding of variables for the subject position in order to generate a binding for the object position or vice versa. Overall a triple pattern can generate and can consume bindings for variables. Things are slightly different as it comes to FILTERs in SPARQL. SPARQL FILTERs restrict solutions to those for which the filter expression evaluates to `TRUE` [8]. So a FILTER can only consume bindings for the variables, but can not generate any bindings. In order to handle this, we have to slightly adapt our present approach. While generating the query graph, the approach stays the same, so for every filter in the query, we add a new node to the query graph next to the nodes for every triple. In order to distinguish the node for a triple and the node for a filter, the node for a filter is drawn with a bold line. For a better differentiation, we also name the node for the filter consuming node. The meaning of the edges stays the same. Consider the following SPARQL query, which is the same SPARQL query as in Listing 1 extended with one FILTER expression:

```
SELECT *
WHERE {
    ?p teacherOf ?c.                  #t1
    ?s takesCourse ?c.                #t2
    ?s advisor ?p.                    #t3
    ?s memberOf ?d.                   #t4
    ?d name 'Department0'.            #t5
    FILTER(?p !=
        <http://www.Department0.
        University0.edu/
        AssistantProfessor4> )        #t6
}
```

Listing 4: extended SPARQL query from Listing 1

The corresponding query graph is shown in Figure 4.

While transforming the query graph from an undirected graph into a directed graph, we use the same approach as discussed before. The only change regards the handling of these consuming nodes. From a naive standpoint of view,

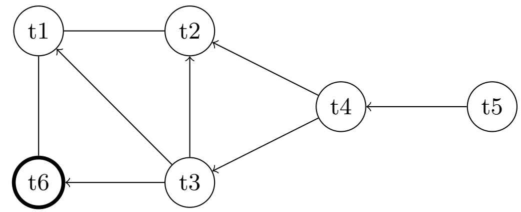

Figure 4: Query Graph for query in Listing 4 after processing the nodes `t5, t4` and `t3`

we can imagine that in general, it can not result in the very best execution time if the filter is placed at an arbitrary position. Based on the experiments in section 4 we were able to substantiate the assumption to place the filter **after** all triples, which have at least one common variable with the filter condition, were processed. Based on the query graph, all edges of the consuming node have to be directed in order to be the next node to be processed.

Considering our example query graph in Figure 4 we have already processed the nodes `t5, t4` and `t3`. In the current state we have to choose between node `t2`, node `t1` or the consuming node `t6`. As described before, the consuming node can only be a candidate, if all edges of the consuming node were turned into incoming edges in order to get the bindings from all relevant triples. So we have to choose between `t2` and `t1`. This is done using the approach presented before. In doing so, we first choose `t2`, afterwards `t1`. Now all edges of the consuming node are directed and we can add the filter to the order of the so far processed triples. If there would be any undirected edge in the query graph, we just continue using our approach. Overall this handling of a filter in a SPARQL query does not mean we always add the filter at the end of the query. Considering the following filter regarding the variable `?s`:

> FILTER(?s != http://www.Department0. University0.edu/GraduateStudent29)

This filter would be added in the current order after we have processed all triples containing the variable `?s`.

In summary, we have adapted our approach to handle the FILTER construct of a SPARQL query.

3.3 Handling Group Graph Patterns next to FILTER

Next to a FILTER, there are many more possibilities to write a group graph pattern in SPARQL, for example OPTIONAL or SERVICE. Also these elements can consume bindings similar to a FILTER, but considering only the triples contained in a group graph pattern, these triples also generate bindings for each other.

Consider the SPARQL query in Listing 5 with a BGP of three triples and an OPTIONAL clause with two triples. The idea is to handle those queries on different abstraction levels. While reordering the triples which are contained directly in the WHERE clause, the group graph pattern is abstracted into one single node. On this basis, we can consider the triple patterns and group graph patterns inside of the current group graph pattern.

```
SELECT *
WHERE {
    ?s advisor ?p.                      #t1
    ?s memberOf ?d.                     #t2
    ?d name 'Department0'.              #t3
    OPTIONAL{                           #O
        ?s :takesCourse ?c.     #t4
        ?p :teacherOf ?c.       #t5
    }
}
```

Listing 5: SPARQL query with OPTIONAL

During the abstraction, the query graph for the first level contains four nodes. The nodes `t1, t2` and `t3` represent the corresponding triple pattern while the node `O` represents the OPTIONAL clause as shown in Figure 5. Based on this, we consider a query graph containing the triples `t4` and `t5`, which are inside of the OPTIONAL. The incoming edges for the second query graph represent the information about already bound variables from outside the OPTIONAL as shown in Figure 5.

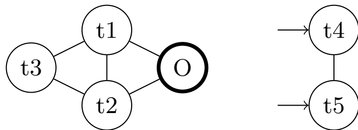

Figure 5: Query Graphs for the query in Listing 5 for the two different abstraction levels

Until now we have only considered queries, which have at most one OPTIONAL group graph pattern. This approach is also applicable if a SPARQL query has more than one group graph pattern. If this is the case, we generate a consuming node for every group graph pattern and use the same approach as described before. We only have to extend the mechanism to handle several consuming nodes as potential next nodes. In order to get the best possible order, we use a preference ranking for the different kind of group graph patterns. So for example, a FILTER is ranked above an OPTIONAL. This approach is quite generic because a SPARQL query can also be nested using an arbitrary number of levels. Conceptually handling these elements like OPTIONAL should be possible using our presented approach as stated before. As described in section 4 we will need more experiments and tests to show the applicability of our approach for queries with group graph patterns like OPTIONAL.

4. EXPERIMENTS

In this section, we present the experiments carried out to test our approach and to get an idea if this approach is promising. We used the LUBM data generator [4] to generate a dataset of approximately 130 000 triples. These triples are stored in *GraphDB*. One of the first experiments was to execute all possible permutations of the BGP of a SPARQL query to get an idea about the different execution times regarding the different orders of the triples. In addition, the query was reordered using our presented approach. For every query tested so far the reordered query was in the forefront of all execution times from the permuted queries. To give an idea how the execution times can differ, the distribution of the execution times regarding all permutations of the query presented in Listing 1 depicted in Figure 6. For this example, our approach achieves the best execution time by on average of 325 ms. All permutations were executed 10 times in order to reach a good average.

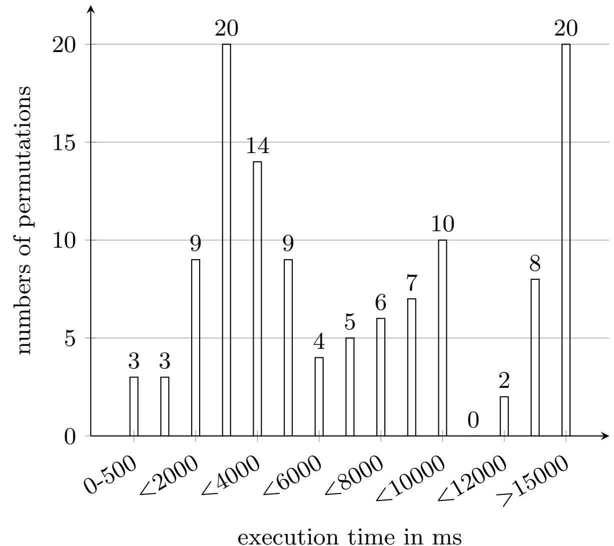

Figure 6: Distribution of the execution times in relation to the permutations of the query in Listing 1

This test setup was executed with several different queries but the overall picture is always the same. In Table 2 we see a comparison between some of the LUBM queries regarding the best possible execution time for one of the permutations compared with the execution time achieved while executing the permutation derived from our approach. In the last column, we listed the percentage of permutations of the respective query, which had the same or smaller execution time than the derived permutation from our approach.

LUBM Query	best execution time	achieved execution time	% of faster permutations
1	11 ms	11 ms	-
2	7 ms	16 ms	0,03056
7	15 ms	15 ms	-
9	4 ms	7 ms	0,07778
12	3 ms	39 ms	0,58334
13	4 ms	4 ms	-

Table 2: Comparism between best possible execution time and achieved execution time using our approach for some LUBM queries

For LUBM query 2 our approach as presented in this paper achieves an execution time of 16 ms, whereas the best permutation achieves an execution time of 7 ms. Based on these two numbers, it seems that our approach does not perform well. If we add the information, that this query has 6 triples, so 720 possible permutations and only approximately 3% of these permutations have the same or better execution time, our approach is in a better position. For some of the queries, as LUBM query 7 with 4 triples, our approach

derived the best possible permutation based on the execution time. These experiments also showed some examples like LUBM query 12, where our approach does not determine a good permutation of the query. These queries will be used to improve our cost model.

Based on the assumption the approach is using a good heuristic, we have also examined the impact of the filter placement. In order to do this, we used our approach to reorder the BGP of the query, added the filter statement at every possible position and compared the execution times again. Also for this test setup, the experiment matches with the expectations about placing the filter after all triples which contain at least one of the variables of the filter. In Table 3 we see the impact of the filter placement.

filter placement	execution time
before t5	289 ms
after t5	279 ms
after t4	281 ms
after t3	277 ms
after t2	277 ms
after t1	274 ms

Table 3: Impact of the FILTER placement for the enlarged SPARQL query in Listing 4

Using our approach for the query as shown in Listing 4 we place the filter after the triple `t5`, which is also the best position according to all possible positions as seen in Table 3. During our experiments, we have observed the impact of the filter condition itself, but explicitly considering the condition of the filter will be part of our future research. Overall our performed experiments showed the applicability of our approach.

5. RELATED WORK

In general, query optimization is a well-established area especially for SQL, but regarding SPARQL queries it is a quite current topic. In the following, we shortly describe some different approaches, which use different methodologies than our approach presented here. The Characteristic Set approach [3] uses dynamic programming based algorithm on a precomputed hierarchical structure, which allows determining the best order of the triples. In [6] they translate a query into a multidimensional vector space and perform distance-based optimization by considering the relative differences between the triple patterns. Comparing different selectivity estimations for a SPARQL query was done in [12], whereby this approach focusses only on BGP queries. It describes different ways to compute heuristics for the optimization but does not consider the structure of the triple. Our approach fills this gap between using statistical data and taking the structure of the query as well as the triple into account.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented our approach for optimizing a SPARQL query by means of using the sideways information passing. We show how to use our approach for a SPARQL query only consisting of a basic graph pattern. Based on this we showed how to adapt the approach in order to handle queries, which also contain FILTERs or other group graph patterns. The experiments based on the LUBM dataset showed the impact on the execution time based on the order of the triples. Also, these experiments functioned as a preliminary proof of concept for our approach. While testing the execution time of all possible permutations of the triples, the order of triples, our approaches chooses, was in the forefront of the smallest execution times.

As described in the title, this is a work in progress paper. Therefore these experiments are not the top of the flagpole. The next steps will include tests with bigger datasets generated with the LUBM dataset generator as well as tests using data and test queries from the *Berlin SPARQL Benchmark (BSBM)* [2]. While the test queries from LUBM are quite short, the queries tested in BSBM contain more triples and also more complex patterns like FILTERs or OPTIONAL clauses. Another benchmark which provides more complex test queries than LUBM is the SP^2Bench [9], which we want to use as a source for test queries.

In order to improve the reordering based on the current approach we want to take into account semantic information. This will be part of our future work.

7. REFERENCES

[1] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, pages 269–284, New York, NY, USA, 1987. ACM.

[2] C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5:1–24, 2009.

[3] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in sparql queries. In *EDBT*, pages 439–450. OpenProceedings.org, 2014.

[4] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[5] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, 1984.

[6] M. Meimaris and G. Papastefanatos. Distance-based triple reordering for SPARQL query optimization. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1559–1562. IEEE Computer Society, 2017.

[7] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.

[8] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[9] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A SPARQL performance benchmark. *CoRR*, abs/0806.4627, 2008.

[10] S. Sippu and E. Soisalon-Soininen. Multiple sip strategies and bottom-up adorning in logic query optimization. In *ICDT*, 1990.

[11] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, Aug. 1997.

[12] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.