Distance-Based Triple Reordering for SPARQL Query Optimization

Marios Meimaris
ATHENA Research Center
m.meimaris@imis.athena-innovation.gr

George Papastefanatos
ATHENA Research Center
gpapas@imis.athena-innovation.gr

***Abstract*—SPARQL query optimization relies on the design and execution of query plans that involve reordering triple patterns, in the hopes of minimizing cardinality of intermediate results. In practice, this is not always effective, as many existing systems succeed in certain types of query patterns and fail in others. This kind of trade-off is often a derivative of the algorithms behind query planning. In this paper, we introduce a novel join reordering approach that translates a query into a multidimensional vector space and performs distance-based optimization by taking into account the relative differences between the triple patterns. Preliminary experiments on synthetic data show that our algorithm consistently outperforms established methodologies, providing better plans for many different types of query patterns.**

I. Introduction

A. Motivation

SPARQL is a W3C recommendation for querying graph data expressed in the Resource Description Framework (RDF). SPARQL query engines are implemented on either native RDF stores, or physical implementations of other paradigms, such as relational databases. In either case, query optimization involves a layer of logical optimization, where the query plan is constructed. Logical optimization of SPARQL queries heavily depends on the ability of a query optimizer to provide good plans. This involves reordering of a query's triple patterns, in order to minimize intermediate results.

Optimizers use statistics and heuristics in order to search the vast space of potential query plans and reach a solution without trading off much pre-processing time for actual query execution time. They rely on first-level statistics, such as the number of distinct triples with a particular property, and assumptions on the values of deeper level statistics (e.g. join cardinalities). In practice, this is not always efficient, because the bias imposed by these techniques gives rise to optimizers that perform very well for particular query patterns, but poorly on others. As an example, consider the following query, where the triple patterns are labelled on the left as $t_1 \dots t_5$:

Listing 1: Example query for the LUBM[1] dataset.

```
    SELECT ?X ?Y
    WHERE {
t1: ?X rdf:type ub:Student .
t2: ?Y rdf:type ub:Department .
t3: ?X ub:memberOf ?Y .
t4: ?Y ub:subOrganizationOf <http://www.
    University0.edu> .
t5: ?X ub:emailAddress ?Z}
```

Assume that the query processor evaluates the query in the order t_1, t_5, t_3, t_2, t_4, relying on statistics that the total number of *Students* is 99k. It will first attempt to match ?X to all 99k *Students* and their *email addresses*, and then retrieve all *ub:memberOf* associations for these. Most of the resulting triples will be discarded in the evaluation of t_2 and t_4, because we are only interested in *Students* that are members of a *Department* that is a *subOrganization* of *University0*. On the other hand, if the order is t_2, t_4, t_3, t_1, t_5, then the triple pattern t_3 will not be evaluated on all *Students*, but only on the members of the matched results for *?Y*. Hence, the evaluation of t_1 and t_5 will be performed on a much smaller set of intermediate results, thus optimizing query answering. This kind of optimization can yield results that are orders of magnitude faster than in the case where no such optimization takes place [2], [3].

In this paper, we introduce and discuss a novel statistical approach for reordering triple patterns in SPARQL queries, which searches a quadratic plan space and builds a plan on a bottom-up manner by taking into account relative differences between triple patterns. This method generates binary left-deep trees, and our evaluation is centered around this sub-problem. Preliminary experiments show that it outperforms traditional statistics- and heuristics- based approaches in query planning.

B. Related Work

There exists a large body of literature on SPARQL query optimization, where many approaches operate under the data independence assumption, which states that data are uncorrelated, and cardinalities of joined patterns are computed using heuristic, cross-product approaches. Stocker et al [4] propose several statistical and heuristic-based planning algorithms that involve cardinality estimation both with and without joins. Neumann and Weikum [5] in the RDF-3X store use a Dynamic Programming algorithm and cardinality histograms. However, the optimizer sometimes spends more time than the actual query execution, some times orders of magnitude slower [3]. TripleBit [6] uses a Dynamic Query Plan Generation Algorithm (DQPGA) for queries with multiple consecutive joins, which can be costly for pattern-rich queries. Gubichev and Neumann provide better estimates by extending Characteristic Sets [3]. Kalayci et al [7] use ant-colonization algorithms for dynamic optimization. The latest version of Virtuoso uses a greedy query optimization approach that is further assisted by dynamic sampling and a combination of full and partial indexes [8]. Finally, Tsialiamanis et al[9] rely solely on triple pattern heuristics to reach good plans without the need for statistics and indexing.

II. Distance-Based Reordering

A. Query Representation

Let T be the set of all triple patterns in an incoming query q. Each triple pattern $t_i \in T$ consists of three nodes, either bound (URIs, blank node IDs or literals) or unbound (variable), i.e., the subject, predicate and object of t_i. Furthermore, let M represent the set of all unique nodes in the subject and object positions in T, both bound and unbound. We map each triple pattern t_i from T to a $|M|$-dimensional vector space, where each vector attribute is an element from M, i.e., a subject or object node from the query. In this preliminary approach, we assume a solution space of left-deep trees, where the original problem of finding the best order of triple evaluation is factorial. This is because for a query q consisting of $|T|$ triple patterns, there are $|T|!$ different permutations of triples. The factorial problem, commonly solved by dynamic programming algorithms, guarantees that the solution will be optimal with respect to the chosen cost model. Unfortunately, recent studies have suggested that even industrial-level query optimizers do not yield satisfactory results when they are heavily dependent on the cost model [10], especially when cost model errors propagate through multiple joins.

In our approach, we map triple patterns from a query into a multidimensional space, and decide the join order based on the spatial correlations of the patterns, i.e., by comparing the pair-wise distances of the vectors that represent the triple patterns in the multiple dimensions, and ranking the patterns based on this comparison. The multidimensional space is built based on cardinality statistics that are held in a separate index. These pre-computed statistics are easily calculated in RDF stores, and include the cardinality of triples with a bound subject, property, and object, as well as the size of the whole dataset. Specifically, we assume that each triple pattern becomes a m-dimensional vector, and the value of an attribute m_i for a given triple pattern $t = (s, p, o)$, is given by the following function:

$$f(t, m_i) = \begin{cases} card(t), & \text{if } m_i \in (s, o) \\ 0, & \text{otherwise} \end{cases}$$

Where $card(t)$ is the cardinality of the triple pattern. This is calculated using the following rules:

- $card(t) = card(p)$, if p is bound and s,o are unbound,
- $card(t) = max(1, \frac{card(p)}{|S|})$, if p and o are bound, and s is unbound,
- $card(t) = 1$, if p and s are bound, and o is either bound or unbound,
- $card(t) = |D|$, for all other cases

The cardinality for a bound property $card(p)$ is simply given as the number of triples with p as the predicate. $|D|$ and $|S|$ are the number of triples in the dataset D, and the number of distinct subject nodes S respectively. Notice that we do not represent predicate attributes, which means that the query space will be built entirely based on the subject and object nodes of all triple patterns in the query. Instead, the information that comes from the predicates in the triple patterns becomes quantized in the values of the attributes, as is given by the aforementioned cardinality estimation rules. Especially for the *rdf:type* property, we assume the existence of an aggregate index that holds the exact cardinalities of particular class types. Therefore, in this special case, the second rule is altered to reflect the exact enumeration of subjects with a specific type. For instance, in our running example, the cardinality of t_1 is 99k, as there exist 99k subjects with *rdf:type ub:Student*.

Function $f(t, m)$ can be used to calculate the values of a $|T| \times |M|$ matrix, denoted as Q_m, where each row represents a triple pattern in T, and each column represents an attribute in M, or simply a node in the query pattern. By applying a distance function and processing the pair-wise distances between rows in Q_m, the search space becomes quadratic with respect to $|T|$, and the comparisons will be of $O(|T|^2)$ complexity, rather than the original factorial one. Moreover, as we do not evaluate self-distances and permuations of the same pair (e.g., $[t_1, t_2]$ and $[t_2, t_1]$, the actual number of comparisons will be $\binom{|T|}{2}$, or $|T| \times (|T| - 1)/2$. Even though this reduction in cost comes with the loss of guarantee of optimality, our experiments show that this approach tends to work well for all types of queries, even with large numbers of triple patterns.

The matrix Q_m for the query of Listing 1, is shown in Table 1 for the synthetic dataset LUBM10 with 1.5m triples. To construct this for dataset D, we apply the estimation rules based on the pre-computed statistics. For instance, there are 99k triples with *rdf:type* as a property and *ub:Student* as an object, which can be seen in columns 1 and 2 for t_1. Similarly, there are 106k triples with *ub:emailAddress* as a predicate, which is encoded in t_5. Application of a distance function will generate a $|5| \times |5|$ distance matrix of pair-wise distances between the triple patterns $t_1 - t_5$.

B. Plan Generation

When Q_m is constructed and the distance matrix is calculated, we build sub-plans for the query, based on the ascending ranking of the pair-wise distances of the triple patterns in T. The algorithm to build sub-plans can be seen in Algorithm 1. Specifically, we sort pair-wise distances in ascending order and create an empty queue for sub-plans (Lines 1-3). Then, we iterate through each pair of sorted triple patterns t_a, t_b (Line 4). If there exists a sub-plan p_i that contains only t_a or only t_b, we append the non-contained triple pattern to p_i (Lines 5-10). If none of t_a, t_b are contained in a sub-plan, we create a new sub-plan and add t_a, t_b, then push the new sub-plan to the existing queue. The order in which we add these two depends on the cost of each triple pattern (Lines 11-15). In case both patterns t_a and t_b exist in sub-plans, we continue to the next pair.

After we have created a series of (ordered) sub-plans, we iterate through consecutive sub-plans and attempt to reorder their patterns in order to better capture join relationships that occur between these. More accurately, if there are more than one sub-plans found, then we try to rearrange the triples that share common variables between adjacent sub-plans, so that the joined triple patterns between the two sub-plans are closely located. This can be seen in Algorithm 2.

Specifically, if there is only one sub-plan, then this is returned as the final plan (Lines 1-3). Else, the algorithm iterates through the $subPlans$ queue with two pointers (Line 4), and prioritizes each pair of sub-plans (Line 5). The $prioritizePair$ function checks for joins between two consecutive sub-plans (Line 12), i.e. it checks whether the last triple (tail) in p_i is joined with the first triple (head) in p_{i+1}. If they are joined, it returns the pair as is (Lines 13-15). If not, it finds the first triple in p_{i+1} that can be joined with the tail of p_i (if one exists), pushes it at the tail of p_i and recursively checks the same pair (Lines 17-21). Finally, the $prioritizePair$ function returns an array of two plans, the first of which is inserted in the final plan, while the second is used as the first sub-plan of the next iteration, as it holds the updated ordering. This means that the triple patterns that are ranked lower in p_{i+1} will be bubbled-up and re-ranked in order to reflect the join relationship between p_i and p_{i+1}.

Algorithm 1 *generateSubPlans*

Input: map**:** A matrix of triple patterns from the query as rows, and query nodes as columns

Output: $subPlans$**:** An ordered list of sub-sets of query triple patterns, that constitute subplans.

```
1:  $subPlans \leftarrow newQueue()$
2:  $distances \leftarrow distanceMatrix(map)$
3:  sort($distances$)
4:  for each $pair\ t_i, t_j \in distances \mid t_i \neq t_j$ do
5:      if $\exists p_i \in subPlans \mid t_a \in p_i$ AND $t_b \notin p_i$ then
6:          append $t_b$ to $p_i$
7:          continue
8:      else if $\exists p_i \in subPlans \mid t_a \notin p_i$ AND $t_b \in p_i$ then
9:          append $t_a$ to $p_i$
10:         continue
11:     else if $\nexists p_i \in subPlans \mid t_a \in p_i$ OR $t_b \in p_i$ then
12:         $p \leftarrow newList()$
13:         append minCost($t_a, t_b$) to $p$
14:         append maxCost($t_a, t_b$) to $p$
15:         $subPlans$.push($p$)
16:     else
17:         continue
18:
    return $subPlans$
```

C. Experiments

We implemented a proof-of-concept version of the query planner in Jena ARQ, and conducted experiments on Jena TDB[1], comparing against several statistical reordering approaches, namely Stocker et al's PFJ and ONS [4], Kalayci et al's Ant System [7] as well as Jena TDB's Fixed (JF) and Weighted (JW) optimizers, and the reordering performed by the open source edition of Openlink Virtuoso 7.2. As triple pattern ordering is a problem that is orthogonal to the low-level implementation specifics and design choices (e.g., join implementations, indexing), we use Jena TDB as a common testbed for all approaches to strictly assess and compare the effect of triple pattern orderings on query processing. We used LUBM[1] to generate a synthetic dataset of 15m triples, and measured execution times for 29 queries on the LUBM dataset. The queries consist of 14 original queries provided by LUBM, and an additional 15 queries of increasingly complex shapes and sizes, used in [7]. All code and queries are available online[2]. For each approach, we extract an ordering, and feed that directly to the Jena query processor. Table 2 summarizes the percentages of best plans for all queries. These are computed by measuring, for each approach, how many plans were the fastest[3] in relation to the other approaches, for all queries. Because of the fact that the fastest plan can be reached by more than one method, we differentiate between

[1]https://jena.apache.org/documentation/tdb/

[2]https://github.com/mmeimaris/sparqlDistancePlanner

[3]We measure execution times by executing each query 10 times and reporting the mean.

Algorithm 2 *reorderSubPlans*

Input: $subPlans(p_0, \ldots, p_{n-1})$**:** A queue of sub-plans

Output: $finalPlan$**:** An ordered list of triple patterns, to be executed by the query engine.

```
1:  if $subPlans$.size $== 1$ then
2:      $finalPlan$.push($p_0$)
3:      return $finalPlan$
4:  for each $p_i, p_{i+1} \in subPlans$ do
5:      $nextPair \leftarrow$ prioritizePair($p_i, p_{i+1}$)
6:      $finalPlan$.push($nextPair$[0])
7:      $p_{i+1} \leftarrow nextPair$[1]
8:      if $p_{i+1}$.isEmpty then
9:          $subPlans.remove(p_{i+1})$
    return $finalPlan$
10: procedure PRIORITIZEPAIR($p_i, p_{i+1}$)
11:     $newPair \leftarrow array[2]$
12:     if $p_i.tail$ is joined with $p_{i+1}.head$ then
13:         $newPair$[0]$\leftarrow p_i$
14:         $newPair$[1]$\leftarrow p_{i+1}$
15:         return $newPair$
16:     else
17:         for each $t_k, \in p_{i+1}$ do
18:             if $p_i.tail$ is joined with $t_k$ then
19:                 $p_i.push(t_k)$
20:                 $p_{i+1}.remove(t_k)$
21:                 $newPair \leftarrow$prioritizePair($p_i, p_{i+1}$)
22:         $newPair$[0]$\leftarrow p_i$
23:         $newPair$[1]$\leftarrow p_{i+1}$ return $newPair$
```

plans based on the generated orderings, and then we compare their execution times. As can be seen, our method outperforms all other methods for queries of different patterns, namely the original 14 queries, and the 15 additional queries of star, chain, chain-star, and cyclic patterns, achieving the best plan 90% of the time, with Virtuoso coming second with 66% of its orderings being the best.

In order to assess how our planner performs with respect to different query types, we use 15 extended queries originally constructed in [7]. These represent different query types, and are thus classified into star, chain, cyclic, and chain-star query types, of varying triple pattern sizes. Their original intention was to provide more complex query structures of up to 14 triple patterns for LUBM dataset, because the original queries are limited to 1-6 triple patterns. We measure execution times for each query type separately, and report these in Tables 3, 4, 5, and 6, for star, chain, cyclic, and chain-star queries respectively. For each query, the best ordering is marked with a bold execution time. While we do not show the actual triple orders that were derived by each method, multiple bold values in the same row indicate that the same order has been achieved by more than one approach. Note that small differences in execution times for the same orderings can be attributed to lags imposed by I/O, available CPU resources and other external factors.

For computing the distance matrix from Q_m, we used a simple Euclidean distance function that measuring distances between triple patterns in the M-dimensional space. Overall, the results are encouraging for further pursuing this direction. In fact, in 90% of the queries, our method generated the fastest plan in relation to the other approaches, of whom the best

TABLE I: Q_m matrix for reference query.

	?X	Student	?Y	Dpt	Univ0	?Z
t_1	99k	99k	0	0	0	0
t_2	0	0	189	189	0	0
t_3	106k	0	106k	0	0	0
t_4	0	0	239	0	239	0
t_5	106k	0	0	0	0	106k

TABLE II: Percentage of plans that are best compared to other methods for all queries.

JW	JF	ONS	PFJ	ANT	VIRT	OUR
59%	48%	28%	59%	31%	66%	90%

(VIRT) achieved a rate of 66%, and the rest achieved less than 60%.

TABLE III: Query execution times (seconds) for the star queries.

	JW	JF	ONS	PFJ	ANT	VIRT	OUR
Q1	1.01	1.14	1.07	**0.61**	1.05	0.63	**0.57**
Q2	0.91	0.92	4.71	0.76	0.79	0.74	**0.67**
Q3	0.23	0.27	0.23	0.15	3.62	0.21	**0.13**
Q4	0.17	0.08	0.07	0.07	0.42	**0.05**	**0.03**

TABLE IV: Query execution times (seconds) for the chain queries.

	JW	JF	ONS	PFJ	ANT	VIRT	OUR
Q1	0.30	0.33	0.37	**0.14**	1.09	0.39	**0.14**
Q2	**0.49**	**0.56**	**0.52**	0.77	**0.50**	**0.48**	**0.48**
Q3	322	323	131	292	139	**55**	**53**
Q4	**54**	**53**	**53**	163	209	182	**49**

III. Conclusions and Future Work

Our preliminary results show potential value in a full implementation of a logical query optimizer. As future work, we intend to design a more mature algorithm for the plan generation, which takes into account a mix of dataset statistics, historical queries and graph summarization in order to provide better estimates that lead to more efficient plans. Taking into account join cardinality estimates will also be implemented. Moreover, we plan on experimenting with different distance functions in order to capture distances between triple patterns more accurately. Finally, we intent to perform exhaustive comparisons with large datasets and queries, as well as other state of the art methods, in order to assess the scalability of the approach, as well as its implementation in parallel settings. Towards this end, we will explore how our method can generate flat bushy n-ary join trees (instead of binary left-deep trees) that tend to be more easily parallelized.

Acknowledgements. This work was partially supported by the projects EU H2020 SlideWiki (#688095) and "Computational Science and Technologies: Data, Content and Interaction" (#5002437, EU Regional Development Fund - Greek national funds).

TABLE V: Query execution times (seconds) for the cyclic queries.

	JW	JF	ONS	PFJ	ANT	VIRT	OUR
Q1	0.05	0.07	0.05	**0.04**	**0.04**	**0.04**	**0.04**
Q2	0.25	0.24	0.19	**0.02**	0.20	0.18	**0.02**
Q3	**0.06**	**0.06**	0.31	2.37	0.74	**0.04**	4.50
Q4	1.06	0.94	1.30	0.64	0.81	**0.04**	0.52

TABLE VI: Query execution times (seconds) for the chain-star queries.

	JW	JF	ONS	PFJ	ANT	VIRT	OUR
Q1	**4.16**	**4.16**	**4.15**	7.39	**4.15**	5.15	**4.15**
Q2	5.95	5.92	5.93	5.83	6.67	**0.49**	**0.42**
Q3	6.11	4.98	4.75	214.92	6.37	**2.35**	**2.31**

References

[1] 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*, vol. 3, no. 2, pp. 158–182, 2005.

[2] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 984–994.

[3] A. Gubichev and T. Neumann, "Exploiting the query structure for efficient join ordering in SPARQL queries." in *EDBT*, 2014, pp. 439–450.

[4] 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*. ACM, 2008, pp. 595–604.

[5] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[6] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "Triplebit: a fast and compact system for large scale RDF data," *Proceedings of the VLDB Endowment*, vol. 6, no. 7, pp. 517–528, 2013.

[7] E. G. Kalayci, T. E. Kalayci, and D. Birant, "An ant colony optimisation approach for optimising SPARQL queries by reordering triple patterns," *Information Systems*, vol. 50, pp. 51–68, 2015.

[8] P. Boncz, O. Erling, and M.-D. Pham, "Advances in large-scale RDF data management," in *Linked Open Data–Creating Knowledge Out of Interlinked Data*. Springer, 2014, pp. 21–44.

[9] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz, "Heuristics-based query optimisation for SPARQL," in *Proceedings of the 15th International Conference on Extending Database Technology*. ACM, 2012, pp. 324–335.

[10] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.