Query Optimization for SPARQL Queries in RDF

Shantanu J. Jain
IU Computer Science Department
Bloomington, IN 47405, USA

shajain@indiana.edu

Deepak B. Konidena
IU Computer Science Department
Bloomington, IN 47405, USA

bkoniden@indiana.edu

Prashant R. Singhal
IU Computer Science Department
Bloomington, IN 47405, USA

psinghal@indiana.edu

Navin Reddy
IU Computer Science Department
Bloomington, IN 47405, USA

navreddy@indiana.edu

ABSTRACT

This paper proposes the technique for optimization of SPARQL queries for the RDF data by efficiently performing join order optimization during the query optimization phase. The join order optimization approach proposed here uses A* algorithm to search through the space of plans (including partial plans) for a given query and returns optimal or close to optimal plan. This entails efficient techniques for plan generation, plan comparison and plan elimination. The plans are generated by first building a small (partial) plan with no relation in it and then generating bigger plans by appending more relations to it step by step, till a complete, promising plan is formed. The heuristics used for the A* search considers the cost of the generated plan based on a simulated cost model as well as an estimated cost corresponding to the number of joins not yet incorporated in the plan. The paper also discusses the various criterion considered for the comparison of plans and how it is used to obtain the best possible plan.

The plans generated using A* search algorithm is compared with the plans generated by using the best first search algorithm. The plans are compared based on the quality of the plan generated and the time taken for the generation of the plans for each of the algorithms.

1. INTRODUCTION

The current Web can be characterized as the second generation Web or Web 2.0. The Web 1.0 or first generation web started with the handwritten HTML pages; the Web 2.0 introduced the machine generated and active HTML pages. These web generations were meant for direct human processing (reading, writing and form filling). Web 3.0 also referred to as Semantic Web aims at machine processable information. It aims to enable intelligent services like search agents, information filters etc. Semantic Web would be possible when further levels of interoperability will be established. Such interoperability is facilitated by W3C standardization efforts notably XML/XML Schema and RDF/RDF (Resource Description Framework) Schema. However, RDF data is not very implementation friendly. There are some RDF implementations like JENA[7], RDF-3X[6] which performs well in specific cases. The SPARQL has been standardized for processing RDF data by W3 consortium. However, there is a lot of scope for improvement in RDF query optimization.

The query optimization techniques implemented so far are tailored to the needs of a relational database. The RDF typically poses the challenge of dealing with large number of joins which make the join order highly significant to optimize. The other challenge is posed by the absence of schema, as it hinders the selectivity approximation.

The technique proposed here considers optimal join ordering in the plan generation for the queries. The plans are generated taking into consideration different orders in which the joins can be performed. This increases the search space for the optimizer to retrieve the most optimal plan for the query execution. In order to counter this problem, the A* search algorithm will be used to prune this search space and generate the best possible plan.

The next section discusses the work done previously in this area. The third section discusses the A* search algorithm which is the base technique used for the plan generation. The fourth section discusses the plan generation techniques being employed, followed by the plan elimination and plan comparison. The criterion for evaluating the cost of plans is being discussed in the subsequent section to highlight the heuristics we have used for evaluating the plans. The last section shows the result that has been obtained by comparing the plans generated with the best first search algorithm.

2. RELATED WORK
2.1 Storage Model
Our implementation takes an assumption of using the hexastore[3] storage model. The model stores the data in 6 tables indexed on the 6 permutations of subject predicate and object. This allows fast access to data. This also becomes handy for performing index nested loop joins. Further, it allows a lot of flexibility with retrieving the data, sorted on any column of interest. This helps in performing more number of merge joins where tables are already sorted on join columns. Both these operations are cheaper as compared to nested loop join and if used amply can improve performance.

2.2 Known Approaches
There has been significant research work done so far in the field of join order optimization. This involves the widely used

approach of dynamic programming [1], [2] and [5]. [2] points out Dpccp which generates all possible query evaluation plans including all the bushy plans. Though this approach gives optimal plan, but the process of plan generation has very high time complexity. Further, the bushy plans along with nested loop join result into blocked execution. [2] Considers generation of only left deep plans which execute in pipeline. However, it neglects all the bushy plans some of which can be more efficient and worth evaluating (example merge join and nested index loop join). We propose an approach using A* algorithm that will consider bushy plans, however it would not search the entire search space and efficiently generate close to optimal plan.

3. A* SEARCH STRATEGY

The A* search algorithm is used for searching through the search space by efficiently pruning the space based on some heuristic. The final result generated by the search is close to optimal result without visiting every possible part of the search. We are using a customized way of A* search where only one list structure is maintained.

The A* search starts with the initial state and keep searching the whole search space till the goal state is reached. At any point of time, the A* search uses a list named OPEN list where it keeps records of the states which have not been expanded yet. The A* search uses two heuristics for carrying out the search process. The first heuristic, say g(x), considers the cost of reaching the current state from the start of the search. The second heuristic, say h(x), considers the estimated cost of moving from the current state to the final state. The total cost of the state at any point, say f(x), in the search is the sum of g(x) and h(x). The states in OPEN list are kept sorted based on the cost, thus its first element is always the one with least cost. This state is removed from the list and expanded to give newer states, which are inserted back in the open list maintaining the sort order. During this process the states being added and the states on the OPEN list are compared and those states that are costlier and equivalent to some other state are pruned off. This helps keeping the OPEN list small (important for good performance). These steps are repeated till the state in the front of OPEN list is the goal state.

In the context of the query plan generation, the initial state refers to the null plan where no join is performed. The end or goal state refers to the complete plan for the query evaluation. The intermediate state can be partial plans which are generated by performing some but not all joins present in the query. The heuristic h(x) considers the number of joins in the query which are left to be performed for the calculation. The g(x) heuristic is based on physical operator's cost model which is discussed in later section of the paper.

4. PLAN GENERATION

Our join order optimizer takes into input, a representation of SPARQL query. We have limited these query to have only single values to bind variables of triple in where clause. For a given a SPARQL query we take the individual triple mentioned in the where clause as a relation. These relations can be obtained by directly querying any of the 6 large triple tables. Any two such relations are joinable if they have one or more common variable. The plans are composed of one or more relations. If a plan has more than one relation then all possible joins between these relations is considered to be complete. A goal plan has all the relations specified in the query. Plan also stores a set of all the columns that will be present in it, if it is actually materialized. It also specifies the column on which it is sorted (in case it is). If a plan is produced as join between two smaller plans it contains them as inner and outer plans. If it is produced as an index join it contains a plan and a relation

4.1 Plan Expansion

The plan expansion phase generates bigger plans from small plans step by step. This involve simple plans for index scans and complex plans for joins between the partial plans based on the common join attributes by using different join operators. The join operators that are considered for the plan expansion involve the nested loop join, index loop join, merge join (input relations already sorted) and sort merge join. For generating new plan, the first plan from the sorted OPEN list is taken and expanded to give new bigger plans. These newly generated plans are added to the open list if they are good. The criteria on which they are eliminated is discussed in plan elimination

First a null plan having no relation is generated. Plans corresponding to index scan are generated from a null plan as the first step of search process. This corresponds to the operation of index scan using the bound columns of relation as keys to access data from any of the 6 triple stores. While generating these plans all the possible ways of accessing data is considered. For example – if the value in the object column is bound, then the plan for index scan primarily on object, secondarily on subject and tertiary on predicate (osp) can be considered. Apart from this ops can also be considered. Note these two index gives result sorted on different columns (one one subject and other on predicate). This can be handy in providing the flexibility of having more merge joins.

All possible index loop join plans are generated for the current best plan in front of OPEN list. To do this all the relations not present in the current plan are checked to see if they can be joined with it. If it is than a index loop join plan is constructed for it using the common columnns between the plan and the relation as the index key in one of the 6 triple store. This allows accessing the values directly using the index and reduces the number of I/Os and CPU operation as compared to the nested loop join.

The nested loop join plan, merge join plan and sort merge join plan is generated whenever the current best plan is combined with another plan present in the OPEN list. Two plans are considered to be combinable if they have no relation in common (i.e. there relations are complement of each other) and have some column in common. New plans are generated considering the two input plans as inner and outer and vice versa.

For nested loop join the new plan remains sorted on the outer plans sort column. This is a relatively costly plan in most of the cases. The generation of merge plans requires both the plans to be sorted on common column. The resultant plan is also sorted on the sorted join column. The merge join plans have less cost as compared to nested loops. The sort merge join performs the join on the plans after sorting on the join attributes. There can be more than one join attribute between the two plans on which the join is to be performed. Different sort merge join plans are generated for each join column. i.e. if there are two different join attributes,

then two different plans will be generated by sorting on one of the two join attributes in each of the two plans.

4.2 Plan Elimination

There will be a lot of plans generated by the different join types. This will cause the increase in the number of unexpanded plans in the OPEN list. To prevent the increase in the size of the OPEN list while keeping the good plans in the list, the plan elimination is done whenever a better plan is found. How the plans are compared is discussed next. All the plans that are generated from the same plan are compared with each other first and maximum elimination is done. A lot of elimination happens during this time as we will see in the plan comparison. The small amount of plans left after this process are compared with plans on OPEN list and more elimination is done. The competent plans are only added to the OPEN list out of the generated list. It is also possible that the plans already present in the OPEN list can get eliminated after the comparison with the generated plans. This approach of two phase plan elimination improves performance. Typically the OPEN list is very large and it is desirable to have very less number of plans to be compared with each plan on open list. The first phase of plan elimination helps to achieve this purpose. The Plan Elimination on whole keeps the OPEN list small or manageable, thus helps performance

4.3 Plan Comparison

This section discusses the criterion on which the plans are compared with each other so that the good plans are added to the OPEN list and the incompetent plans are either removed or not added to the list.

The plan comparison takes place between the two plans. In the plan comparison, it is first checked if any of the two plans is a sub-plan for another. If one of them is a sub-plan (contains the same relations or a subset of the relations in other plan) of another, then the cost of both of these plans are compared. This cost of the plan is calculated at the time of plan generation. If the cost of the sub-plan is more than the other, then the sub-plan is immediately discarded unless it is sorted on a column other than the sorted column of super plan. As it might be handy in performing some merge join in future. This case is important for consideration because it is possible that this sub-plan can be helpful in lower cost in later stages of the plan generation. If the cost of the sub-plan is less, then the plan is kept for later use unless it contains the same relations as in the super plan.

5. COST MODEL

As discussed earlier, the A* search uses the heuristic at any point in the search space to make decision about the next state to expand. The state here refers to the plans. The heuristic has two different values $g(x)$ and $h(x)$. The $h(x)$ heuristic considers the number of joins that are left to be performed. This number is multiplied with a weight to calculate the value of $h(x)$. We have taken the value of weight equal to 100. The $g(x)$ heuristic is calculated based on the cost of the plan generated so far. This cost is based on the I/O cost and the CPU cost. The I/O cost is calculated based on the number of pages fetched from the disk and the CPU cost is calculated based on the number of comparisons done on the join operation. The different join operations viz Nested loop, Index scan, Index Loop, Merge and Sort - Merge give different CPU costs. The cost model interacts

with the plan cardinality generator to calculate the cost of each plan. The plan cardinality generator is discussed in the next section.

N = Number of Pages in the inner relation

n = Cardinality of inner relation

M = Number of Pages in the outer relation

m = Cardinality of outer relation

The tables below give the I/O cost and the CPU cost for various join operators:

Nested Loop Join	$M + m * N$
Merge Join	$M + N$
Sort-Merge Join	$M*\log_2 M + N*\log_2 N + M + N$
Index Loop Join	m

IOCost

Nested Loop Join	$m * n$
Merge Join	$m + n$
Sort-Merge Join	$n*\log_2 n + m*\log_2 m + m + n$
Index Loop Join	m

CPUCost

The estimated cost of plan $h(x) = 100 *$ number of relations left to complete the plan. And $g(x)$ for complex plan is equal to sum of the $g(x)$ values of underlying plans and cost of combining the two plans, obtained from cost model.

6. PLAN CARDINALITY GENERATOR

Our project is related to join order optimization that can work only if plan cost can be computed. However, estimating the costs of plans requires good estimates of cardinality of tables formed materializing underlying plans. To simulate these estimates we use a plan cardinality generator that actually executes database queries to materialize each and every plan and store the cardinality thus obtained in a PlanCardinality table. This is a one time process for each query we want to optimize. The next time we optimize these queries it will directly query the PlanCardinality table and retrieve the cardinality of plans to feed it to Cost model. The plans generated for SPARQL queries are parsed down to the individual SQL queries

7. ALTERNATE IMPLEMENTATION MODELS FOR COMPARISON

We have developed two different models for comparison with our approach for plan generation. These two models generate plans based on the join ordering but the search procedure in these two models differs from the one we have used. The cost model used for calculating the goodness of the plan at any time for these models is same as the cost model that we have used for our approach. These two models are described in the next subsections.

7.1 Modified A* Search Model

The modified A* model uses the same idea of A* algorithm we have implemented but it differs in a few respects. This model generates the plans at every search stage and chooses the best plan among all as described earlier in our A* approach. However, it does not prune the search space and therefore, all the generated plans are added to the OPEN list. This leads to an increase in the size of this list. However, it always chooses the most optimal plan

at any stage of the search space. The cost of each of the plans is calculated in the similar manner as it is calculated for our approach.

7.2 Greedy Search Model

The greedy search model works on the technique of selecting the most optimal plan from the newly generated plans at any search stage. It does not have the concept of an OPEN list. However to generate the bushy plans it needs to keep a list of prior plans (not sorted). It starts with null plan generates new plans from it and picks up the most optimal plan from these plans only. The other plans go to the list. The best plan is expanded again combining with the plans on the list. Among these newly generated plans again the best one is selected. This process continues till the complete plan is generated. This model never backtracks and finds the solution by moving forward and picking the best plan found at that particular search stage.

8. Results

In order to compare our approach with the two models described in the previous section, we have chosen two different criteria. The first criterion considers the quality of the plan generated by the various approaches. The cost model returns a numeric value depending on the goodness of the plan. The lower the value, the better the plan will be. The second criterion is the size of the OPEN list. This criterion is chosen since at any stage of plan generation, the plan generated is compared with the plans present in the OPEN list or is combined with them to generate higher plans. Therefore, it is necessary to limit the size of the OPEN list for efficient search and faster results.

The data set that is used for the comparison is the standard Barton [5] dataset and the standard benchmark queries are used for the comparison. Most of the queries in the benchmark have less than 4 joins. To evaluate queries with more joins, we have also used some of our own queries on Barton dataset. There are 5 queries for which we compared the performance of each of these 3 approaches. Tables below show the result of the performance for 5 different queries on cost and on the list size.

Query No.	A* Search	Modified A* search	Greedy Search
1	770	770	770
2	2	2	2
3	1	1	1
4	24	24	24
5	1	1	1

Cost Comparison

Query No.	A* Search	Modified A* Search	Greedy Search
1	1	4	6
2	2	34	36
3	4	33	35
4	1	24	26
5	15	>150	>170

List size Comparison

The result in Table 4 shows the maximum number of plans that were present in the OPEN list at any point of time during the plan generation.

The same cost in the Table 3 for all the models is because of the less number of joins that were involved in the queries chosen for testing. This less number of joins lead to the similar cost for all the three models. All the queries designed by us gave empty results. So the costs obtained by using those cannot be used to evaluate the performance. Table 4 shows the significant gain in the efficiency of using the A* search technique in comparison to the Greedy search model and the modified A* search model.

The plans given by all the three approaches are good. They have bias towards using index nested loop join and merge join. This result is as per our expectations, as these are cheap join operators.

9. CONCLUSION

The results do not show significant gain in the quality of the plans generated. However, we believe that the A* technique can help us in generating better quality plans for the SPARQL queries where there are a lot of joins involved. The results definitely show us that pruning the OPEN list and the search space helps us in maintaining the same optimality of plans as those generated by other techniques even after using the large OPEN list set of plans.

10. REFERENCES

[1] Moerkotte G., Neumann T. 2008. Dynamic Programming strikes back. Proceedings of the 2008 ACM SIGMOD International Conference on Management of data.

[2] Moerkotte Guido. 2006. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In Proc. 32nd International Conference on Very Large Data Bases.

[3] Weiss Cathrin, Karras Panagiotis, Bernstein Abraham. 2008. Hexastore: sextuple indexing for semantic web data management. Proceedings of VLDB Endowment.

[4]Artificial Intelligence a modern approach by Stuart Russell and Peter Norvig Second edition Chapter 4 Page 97

[5] Selinger G. Peter et. al. 1979. Access path selection in the relational database management system. Proceedings of 1979 ACM SIGMOD International Conference on Management of Data.

[6] Neumann T., Weikum G. 2008. RDF-3X: A RISC Style engine for RDF. Proceedings of VLDB Endowment.

[7] Carroll J.J. et. al. 2004. Jena:Implementing the semantic web recommendations. International World Wide Web Conference.