Optimizing SPARQL queries over the Web of Linked Data

B. R. Kuldeep Reddy
Indian Institute of Technology Madras
Chennai, India
brkreddy@cse.iitm.ac.in

P. Sreenivasa Kumar
Indian Institute of Technology Madras
Chennai, India
psk@cse.iitm.ac.in

ABSTRACT

The web of linked data represents a globally distributed dataspace. It can be queried with SPARQL whose execution takes place by asynchronously traversing the RDF links to discover data sources at run-time. However, the optimization of SPARQL queries over the web of data remains a challenge and in this paper we present an approach addressing this problem. The proposed approach works in two-phases to optimize the SPARQL queries. The first phase analyzes the query before its execution and discovers classes of data that do not contribute towards answering it which can then be prevented from being fetched. However, this analysis may miss a number of patterns that can only be discovered at run-time. The second phase analyzes the query execution to discover more patterns which are used to further reduce the amount of data fetched from the web to answer the query. The main idea of this phase is to model the query execution as a context graph that is used by a heuristic to discover the patterns which are passed sideways to prune nodes in the context graph, resulting in an improvement in query performance. The implementation of our approach demonstrates its benefits.

1. INTRODUCTION

Semantic data management includes the various techniques that use data based on its meaning. There has been a recent growth in publishing data on the web following the linked data principles forming a global web of linked data. Efficient querying of this web of linked data is thus an important component of Semantic Data Management and is addressed in this paper.

The traditional World Wide Web has allowed sharing of documents among users on a global scale. The documents are generally represented in HTML, XML formats and are accessed using URL and HTTP protocols creating a global information space. However, in the recent years the web has evolved towards a web of data[3] as the conventional web's data representation sacrifices much of its structure and semantics[1] and the links between documents are not expressive enough to establish the relationship between them. This has lead to the emergence of the global data space known as Linked Data[1]. Linked data basically interconnects pieces of data from different sources utilizing the existing web infrastructure. The data published is machine readable which means it is explicitly defined. Instead of using HTML, linked data uses RDF format to represent data. The connection between data is made by typed statements in RDF which clearly defines the relationship between them resulting in a web of data. The Linked Data Principles outlined by Berners-Lee for publishing data on the web basically suggests using URIs for names of things which are described in RDF format and accessed using HTTP protocol.

The RDF model describes data in the form of subject, predicate and object triples. The subject and object of a triple can be both URIs that each identify an entity, or a URI and a string value respectively. The predicate denotes the relationship between the subject and object, and is also represented by a URI. SPARQL is the query language proposed by W3C recommendation to query RDF data[8]. A SPARQL query basically consists of a set of triple patterns. It can have variables in the subject,object or predicate positions in each of the triple pattern. The solution consists of binding these variables to entities which are related with each other in the RDF model according to the query structure. An execution approach for evaluating SPARQL queries on linked data is proposed in [6]. The idea presented in [5] maintains index structures that are used to select relevant sources for query answering, as in federated query processing. However, the index covering large amount of data on the Web requires much more resources. According to [6] a SPARQL query is basically executed by iteratively dereferencing URIs to fetch their RDF descriptions from the web and building solutions from the retrieved data. It must have a seed URI as the subject of the first query pattern. It is explained with an example below.

```
SELECT ?friend ?entity ?number WHERE
{
 <http://site/Tim_lee.rdf> nm1:hasFriend ?friend.
 ?friend nm1:worksFor ?entity.
 ?entity univ:numberOfStudents ?number.
}
```

Figure 1: Example SPARQL query

Example. The SPARQL query shown in Figure 1 searches for friends of Tim-Lee who work for an entity and the num-

Workshop on Semantic Data Management (SemData@VLDB) 2010, September 17, 2010, Singapore.

ber of students in that entity. The query execution begins by fetching the RDF description of Tim-Lee by dereferencing his URI. The fetched RDF description is then parsed to gather a list of his friends. Parsing is done by looking for triples that match the first pattern in the query. The object URIs in the matched triples form the list of Tim-Lee's friends. Lets say `<http://site1/John.rdf>`, `<http://site2/Peter.rdf>`, `<http://site3/Mary.rdf>` were found to be his friends. The query execution proceeds by fetching the RDF descriptions corresponding to John,Peter and Mary. Lets say first Mary's graph is retrieved. It is parsed to check for triples matching the second query pattern and it is found that Mary works for a university `<http://site4/universityA.rdf>`. The university's details are again fetched and the third triple pattern in the query is searched in the graph to get the number of students. Peter's and John's graphs and their companies details would also be retrieved and the query execution proceeded in a way similar to Mary's. But since Peter and John work for corporate entities which do not have students, the third triple pattern is not answered for them.

The SPARQL query shown in Figure 1 can be optimized as follows. Lets say Tim-Lee has friends who are either professors or managers. Suppose Mary is a professor and Peter and John are managers. The professor works for a university and managers for a corporate entity. The third triple pattern in the query can only be answered if the entity is a university as only a university has students. Fetching the details of managers and the companies they work for does not result in the third triple pattern being answered and therefore they need not be fetched. The decision on whether to continue the execution for a friend can be taken immediately when his RDF description is fetched thus avoiding retrieving his company information in-case he is a manager. In this example when John's,Peter's and Mary's data is fetched it is determined that only Mary is a professor and the execution proceeds only for her resulting in performance benefits.

The proposed approach works in two phases. The first phase analyzes the query with the help of vocabularies(RDFS or OWL) used to describe resources involved. This phase traverses the query graph annotating nodes with the classes of data they can have. It determines the classes of data that do not produce results for the query and thus helps in its optimization. However, it is limited in the type of patterns it can detect and hence calls for an additional run-time phase. The second phase models the query execution as a context graph and employs a run-time heuristic that discovers patterns. Many adaptive techniques have been proposed [2] that operate at run-time. Our method is similar to the sideways information passing technique [7] in which the discovered patterns are passed along sideways during query execution. These patterns are then used to avoid retrieval of data by pruning nodes in the context graph, similar to branch and bound technique. The heuristic is effective in optimizing the query but it comes at the cost of completeness of results. Some patterns may be precise like *a teaching Assistant of a course will not credit it* which does not affect the completeness of results. But others like the *faculty who publish a paper with a student is his advisor* are an approximation and its use will not give complete results.

2. QUERY ANALYSIS PHASE

The query graph is divided into different levels according to its structure. A level is associated with each triple pattern in the query. The level of a node in the query graph can also be viewed as its distance from the root in terms of number of predicates. Before the query execution can begin, we analyze it with the help of vocabularies used to describe resources in this phase(we assume a mapping exists that maps elements from one vocabulary onto another as according to [1] it is considered good practice to do so). The idea is to identify classes of data which do not contribute during the query execution in that they do not produce results. The data corresponding to those classes is deemed redundant and may not be fetched from the web. The vocabulary contains the domain and range values for the predicates as well as the subclass hierarchy details. This information can be exploited to restrict the query execution to only those classes which generate results.

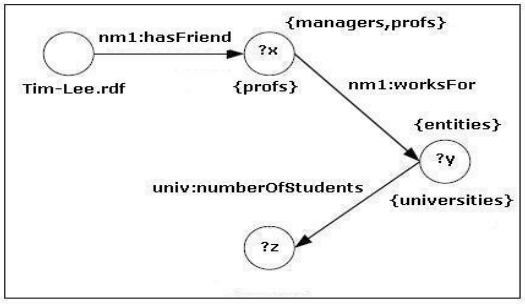

Figure 2: Example Query Graph

Example. Let us consider the query of Figure 1 represented as a graph in Figure 2. The first triple pattern's predicate is *hasFriend* whose vocabulary description reveals that its range is the class of people who can be either managers or professors. The second triple pattern's predicate is *worksFor* whose domain is again the class of people and range the class of entities found from the vocabulary. The third triple pattern's predicate is *numberOfStudents* whose domain is the class of universities. We assume that there exists a mapping which is able to map elements like professors and universities classes and worksFor and employs predicates between the two vocabularies in the query. The third node is therefore restricted to only the class of universities. The vocabulary also reveals that universities employ professors. This information is then used to restrict the second node to only the class of professors. Thus, using vocabulary of the resources involved we are able to detect the pattern that managers do not work for universities. Usage of this pattern reduces the amount of data fetched from the web optimizing the query.

Algorithm. The query is represented as a directed graph and the algorithm basically works by traversing it. It begins by dereferencing URIs of all the predicates in the query. The description of the predicates in the vocabulary give their domains and ranges. The links corresponding to the domains and ranges are also resolved and their details also fetched to discover further information regarding their subclasses. This is taken care of in lines 1-4 in Algorithm 1. The query analysis begins at the root's(seed URI) predicate ,$predicate^0$, and proceeds by taking one outgoing path ,$predicate^i$, at a time in a breadth first manner in lines 7-16. Once a path is taken, its object node is annotated with the classes of its predicate's

range. In lines 11-13 all the domain classes of the paths from this object node are noted and their intersection taken. If it is a subclass of the current class, it replaces the current class of object node in lines 14-15. Now that the node has been restricted, its ancestors are checked whether they also can be restricted with this new information in line 16. For example, if the node has been constrained to the class of universities and its parent node has the class of managers and professors. If there is schema element which says university employs professors then we can restrict its parent to the class of professors. This procedure is repeated till all the predicates are traversed. The final classes annotated on the nodes, indicate that the query execution be restricted only to them resulting in lesser data being fetched from web.

Algorithm 1: Query Graph Analysis

Input : :Query graph Q
Output: :Updated node annotations of Q with classes indicating that the query execution be restricted to them

1 **foreach** *Predicate P of Q* **do**
2 Resolve P to fetch its description R
3 parse R to find out P_{domain} and P_{range}
4 further fetch & parse P_{domain} and P_{range} for their subclasses hierarchy
5 $list \leftarrow null$
6 append $predicate^0$ into $list$
7 **while** *list is not empty* **do**
8 remove $list_{head}$ & assign it to p
9 $node \leftarrow p_{object}$; $node_{classes} \leftarrow p_{range}$
10 $domain \leftarrow universalSet$
11 **foreach** *outgoing* $predicate^i$ *from node* **do**
12 $domain \leftarrow domain \cap predicate^i_{domain}$
13 append $predicate^i$ to $list$
14 **if** $domain \subset p_{range}$ **then**
15 $node_{classes} \leftarrow domain$
16 restrict $node_{ancestors}$

Though this phase serves to identify classes of data which do no produce results, it is limited in the patterns it can detect. Suppose we are given a query to find people in a department who take a graduate course. Following the above algorithm the range of first triple pattern predicate *people* is the class of faculty and students. The domain of the second triple pattern predicate *takesCourse* is the class of students. Therefore the node connecting both the triples is constrained to the class of students. But the third triple pattern restricts the type of courses to *graduateCourses*. Since, we do not have information in the vocabulary which says Graduate courses are generally only taken by students who have advisors we are unable to further constrain the node identifying the people from students to only students who have advisors. This calls for a run-time approach which analyzes the query execution to discover patterns which are missed by the query analysis phase.

3. QUERY EXECUTION PHASE

3.1 Data-Structures

3.1.1 Context Graph

This phase models the query execution as a context graph G. The context graph is built by adding nodes to it containing the RDF descriptions of the entities fetched from the web. A node is added as a child to the one containing the RDF description that led to it being fetched. The edge between the two nodes in G is labeled with the predicate name denoting the relationship between the two entities in them. The root is the seed which begins the execution of the query. The results occur at the leaves whose path to the root matches their corresponding sequence of predicate terms in the query. The context graph is also divided into different levels similar to the query graph. The context graph contains bindings of the nodes in the query graph. If the subject binding of a triple pattern in the context graph belongs to level l, the object binding belongs to level $l + 1$. The contents of a node of context graph contains the description of the entity in the shape of a star graph. The center represents the entity being described and the edges denote its properties and their values. The Context Graph also acts as a cache. This allows us to check whether a resource's details have already been fetched earlier before an attempt is made to retrieve it from the web, which further improves the query performance.

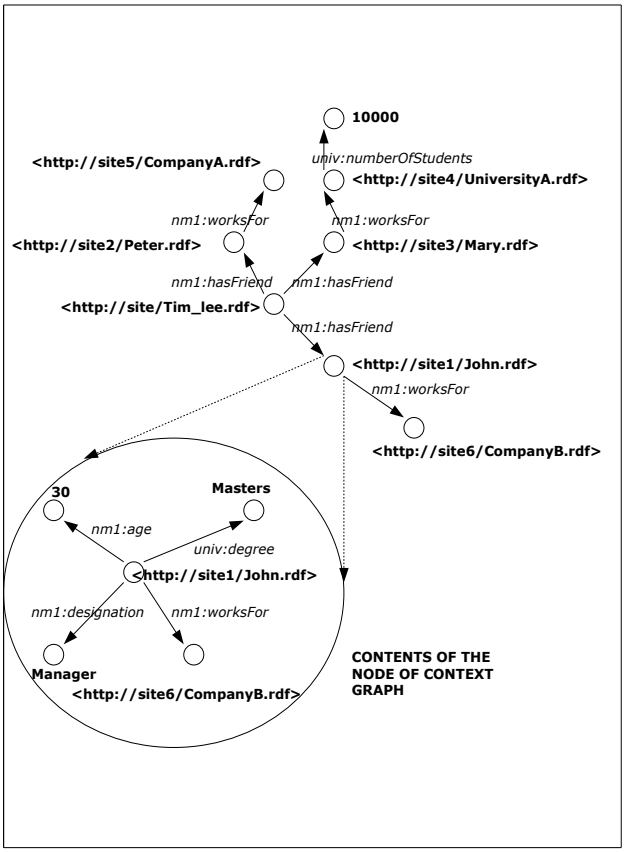

Figure 3: Context Graph

Example. An example for a Context Graph is shown in Figure 3 that models the execution of the query in Figure 1. According to the query, Tim-Lee's URI is resolved to fetch his RDF description from the web and is stored in the corresponding node in the context graph. The context graph is built by creating nodes for Peter,Mary and John

after fetching their details. It continues till the data for the last query pattern is retrieved. The results in this case is the node containing the value 10000 and its path to the root.

3.1.2 Summaries

Summaries are maintained of context nodes which are used by the heuristic to make a decision on whether to explore the current node of the context graph or not. Two summaries are associated with each node of the query graph. One summary $S^{qnode}_{results}$ - summarizes the information of all the result producing context nodes associated with *qnode* and the other $S^{qnode}_{noResults}$ - summarizes the information of all the nodes that did not lead to results for *qnode*. A summary is basically a star graph whose center represents a generic result producing or a non-result producing entity and edges denote the properties. The edges have a weight indicating the number of times that the triple containing that predicate and object has occurred in the nodes previously fetched. An edge with a high weight in result-producing summary and low weight in the non-result producing summary can be used as a discriminant. If a node contains such an edge then it is highly probable it will produce results as it is similar to the nodes earlier that did produce results. Summaries at any point of time contain only the top-K high weight triples.

Algorithm. When a URI is resolved to fetch a RDF description from the web, it is inserted in the context graph as a new node N. Lets say it is associated with node *qn* of the query graph having the two summaries associated with it. If the execution continues for the new node and it produces results, its details are inserted in $S^{qn}_{results}$ else they are inserted in $S^{qn}_{noResults}$. The insertion procedure is as follows. Each triple in the new node N is compared with the triples in one of the summary based on whether it produced results. If both the predicate name and the object match, weight of the corresponding edge is incremented by one. Otherwise, if the number of triples in the summary is less than K, it is inserted as a new triple and its weight is set to one. On the other hand, if the number of triples are greater than K it can randomly select the triple with least weight and replace it. This executes for all the nodes constituting the path from N to the leaf updating summaries at all the levels with the corresponding details of nodes.

3.1.3 Links

The query graph may have a number of nodes which have relationships amongst them other than those specified by the query. These relationships are discovered by analyzing the execution of the query. Their discovery improves the query performance by introducing an edge denoting the relationship between the query nodes, bypassing the original edges and nodes if none of the bypassed nodes need to be displayed. Therefore, the data corresponding to the bypassed nodes need not be fetched thereby reducing the query execution time.

Example. Suppose we have a query which finds out all the students who have authored a paper with a faculty member. In the query graph the nodes denoting students and faculty has a node representing publications in between. It is executed by fetching the details of students followed by their publication details followed by its co-authors to find a faculty member. During its execution we find such a faculty member is generally an advisor of the student. Therefore, using this pattern we can check whether the student has a publication and if he does, display the advisor without fetching the details of publications.

Summaries by themselves are unable to discover these relationships which calls for additional data structures. Therefore two links structures are maintained between each pair of nodes, say $n1$ and $n2$ in the given query graph. The two types of link structures between the pair of nodes in the query graph at level l are $link^{n1-n2}_{results}$ and $link^{n1-n2}_{noResults}$. They are again star graphs with the center indicating a generic result or non-result producing node with weighted predicates attached to it. They are maintained as follows. If the result for a query is produced, the predicate denoting the relationship between node bindings of $n1$ and $n2$ is inserted in the star graph $link^{n1-n2}_{results}$ and its weight set to 1. If the predicate is already present, its weight is incremented by one. If the result for the query is not produced $link^{n1-n2}_{noResults}$ is similarly updated. This process is carried out for all pairs of nodes in the query graph. The bypassing of nodes under certain circumstances may not be correct as it may lead to results which would not have passed through the bypassed nodes. Therefore, to deal with this situation, when a result is generated after bypassing we randomly check whether it would have also been generated by the un-bypassed original nodes. A variable is maintained which continuously computes the percentage difference of produced results between these two situations. If it is falls below a certain threshold, we say that bypassing of nodes is not correct and we drop it.

3.2 Heuristic

We can optimize the SPARQL query by halting the exploration of certain nodes in the context graph which do not produce results. This requires the formulation of a heuristic which makes such a decision. This is similar to the Branch & Bound strategy using a heuristic to prune nodes in the context graph. Reduced number of nodes corresponds to lesser number of URIs being dereferenced to fetch their RDF descriptions and thus lesser time taken to execute the query. The heuristic basically works by matching the contents of a node with the contents of the summaries and links associated with all nodes of the query graph. The summaries are compared to discover edges which have a high weight in either one of them. Presence of such edges in the node are used to assess its likelihood of producing results. Links are used to discover relationships between query nodes not specified in the query graph.

Example. For the query in Figure 1, suppose Tim-Lee has another friend Richard who is a professor in some university. When his RDF description is fetched, a node is created for him in the context graph in Figure 3 in level 1. In order to decide whether to continue exploring with the current node, it is matched with the two summaries associated with the query node "?friend" at the same level. Comparison of two summaries indicates that result producing nodes contain the triple defining the entity to be type Professor and non-result producing nodes contain the triple defining the entity of type Manager. Richard's description is found to contain the triple describing him as a professor and therefore it is predicted that his node will generate results.

Algorithm. The heuristic is invoked each time a new node N is inserted into the context graph. The contents of the new node is a description of an entity in the form a star graph. Lines 2-11 in Algorithm 2 iterate over all

the nodes in the query graph. The idea here is to find if N contains a predicate which denotes a direct relationship between nodes in the query graph allowing us to bypass original nodes between them. The heuristic iterates over all the predicates in N in line 3 and counts its matches in $link_{results}$ and $link_{noResults}$ and stores the scores in variables $scoreL_r$ and $scoreL_{nR}$ respectively in lines 6 and 9. If the ratio of $scoreLr$ to $scoreL_{nR}$ is greater than the threshold, we introduce a direct edge between the two nodes and exploration proceeds with this new edge in lines 10-11.

Lines 12-29 in Algorithm 2 iterate over all the triples in N. Each triple in N is compared with the triples in the two summaries associated with its corresponding query node $n1$ in the query graph. This happens in lines 14-24 in the algorithm 2. Two score variables are maintained for each summary. *scorePO* counts the predicate and object value matches like *x advisorOf studentA* whereas *scoreP* counts the predicate matches like *advisorOf* between the node's and summary's contents. Lines 17-18 and 23-24 do the comparisons and increment the scores by the weights of respective edges. Ratios of the scores computed from result summaries with that computed from non-result summaries are determined. These ratios indicate the presence of edges which are predominant in $S^n_{results}$ but rare in $S^n_{noResults}$. If either of the ratios are above a certain threshold in lines 25-26, we can say that N contains a pattern commonly found in the result producing nodes of its level. Hence, it is predicted to produce results and the execution proceeds from the new node. The inverse of ratios computed earlier are also compared with the threshold in lines 27-28, if they are greater it means that N is very similar to the non-result producing nodes and it may also not produce results. Parameter p represents the probability with which a node may be explored after being rejected by the heuristic. When the node is found to be similar to the non-result producing nodes, the probability of it being explored is diminished by a constant e in line 28.

During the initial phase of the execution, the decisions made for the nodes may not be correct. This happens because the number of nodes to be compared against are less initially. If the current node is of the type which produces the result, it is unlikely that similar type of an entity may have been encountered earlier. The heuristic in this case is unable to determine its result producing capability. To overcome this shortcoming a parameter δ is introduced which allows the exploration of a node even if the heuristic suggests otherwise. This is crucial during the initial part of the execution when the heuristic fails to predict the usefulness of the node. But in the latter part of the execution, the heuristic can bank on a lot of nodes which makes its decision very reliable. This suggests that the parameter vary continuously. It has a high value initially and it decreases continuously with each new node that leads to a result.

Algorithm 2: Heuristic

Input : :Context graph and Query graph, New node N
Output: :Decision on whether to explore N

1 New node N associated with node $n1$ in query graph.
foreach *node n2 in query graph* **do**
2 **foreach** *predicate prd in N* **do**
3 $ScoreL_r \leftarrow 0$; $ScoreL_{nR} \leftarrow 0$
4 **foreach** *Triple* $Tr \in link^{n1-n2}_{results}$ **do**
5 **if** $Tr_{predicate} = prd$ **then**
6 $ScoreL_r = Tr_{weight}$
7 **foreach** *Triple* $Tnr \in link^{n1-n2}_{noResults}$ **do**
8 **if** $Tnr_{predicate} = prd$ **then**
9 $ScoreL_{nR} = Tnr_{weight}$
10 **if** $\frac{scoreL_r}{scoreL_{nR}} >= RATIO$ **then**
11 Explore N with the next predicate *prd* instead of the original path ; exit
12 **foreach** *Triple T in N* **do**
13 $scorePO_{results} \leftarrow 0$; $scoreP_{results} \leftarrow 0$
14 **foreach** *Triple* $Tr \in S^{n1}_{results}$ **do**
15 **if** $Tr_{predicate} = T_{predicate}$ **then**
16 **if** $Tr_{object} = T_{object}$ **then**
17 $scorePO_{results} = Tr_{weight}$
18 $scoreP_{results} = scoreP_{results} + Tr_{weight}$
19 $scorePO_{noResults} \leftarrow 0$; $scoreP_{noResults} \leftarrow 0$
20 **foreach** *Triple* $Tnr \in S^{n1}_{noResults}$ **do**
21 **if** $Tnr_{predicate} = T_{predicate}$ **then**
22 **if** $Tnr_{object} = T_{object}$ **then**
23 $scorePO_{noResults} = Tnr_{weight}$
24 $scoreP_{noResults} = scoreP_{noResults} + Tnr_{weight}$
25 **if** $\frac{scorePO_{results}}{scorePO_{noResults}} >= RATIO$ *or* $\frac{scoreP_{results}}{scoreP_{noResults}} >= RATIO$ **then**
26 Explore N ; exit
27 **if** $\frac{scorePO_{noResults}}{scorePO_{results}} >= RATIO$ *or* $\frac{scoreP_{noResults}}{scoreP_{results}}$ **then**
28 With probability δ/e Explore N and exit
29 Explore N with probability δ

4. EXPERIMENTS

The experiments were conducted on a Pentium 4 machine running windows XP with 1 GB main memory. All the programs were written in Java. The synthetic data used for the simulations was generated with the LUBM benchmark data generator [4]. The LUBM benchmark is basically an university ontology that describes all the constituents of a university like its faculty,courses,students etc. The synthetic data is represented as a web of linked data with 200,890 nodes denoting entities and 500,595 edges denoting the relationships between them. The efficacy of the proposed idea was demonstrated by executing a set of complex queries on the simulated web of linked data of a university and comparing the results with the existing approach. There are a number of patterns in the data and each of the query below demonstrates the ability of the proposed approach to discover and use them to optimize its execution. The results are judged according to the two metrics discussed next.

Metrics. The time taken to execute the query is proportional to the number of URIs resolved to fetch their RDF descriptions during the course of query execution. The time taken to determine whether to explore a node or not is dominated by the amount of time taken to retrieve the data from the web. Therefore, minimizing the number of URIs resolved has a bigger impact on the the query execution time. Thus the results are judged according to two metrics. The first metric α represents the percentage reduction in the number of URIs dereferenced compared with the existing approach, indicating the degree of optimization achieved. The second

metric β denotes the percentage reduction in the completeness of results compared with the existing approach.
let N_e : Number of URIs dereferenced by existing approach
let R_e : Number of results generated by existing approach
let N_p : Number of URIs dereferenced by proposed approach
let R_p : Number of results generated by proposed approach

$$\alpha = \left(\frac{N_e - N_p}{N_e}\right) * 100 \; ; \; \beta = \left(\frac{R_e - R_p}{R_e}\right) * 100$$

Queries

Query1. Return the students in a department who take a graduate course.

The query execution with the existing approach begins by fetching the details of all the students in the department followed by the details of courses taken by them. The courses are then filtered according to whether they are of the type graduate course or not. The proposed approach optimizes the query by recognizing that most of the graduate courses are taken by students who have an advisor. This is used to avoid retrieving course details of students who do not have an advisor.

Query2. Return the students whose advisors have an interest in the research area "Research9".

The existing approach executes the query by fetching the details of all the students followed by the details of their advisors. The advisor's research interests are further retrieved and checked to see if its title is "Research9". The query is optimized by the proposed approach by building a list of faculty involved in "Research9". Then as the students details are fetched, they are checked to see if their advisors belong to the list involved in "Research9". This results in avoiding the retrieval of data regarding advisors and their researching interests not involved in "Research9".

Query3. Return the students who co-authored a paper with a faculty member.

The query is executed by the existing approach by first fetching the details of all the students followed by the details of their publications. The publication details reveals its co-authors whose details are again fetched. If any one of the co-author is a faculty member, his URI is displayed as a result along with the student's. The proposed approach discovers that most of such faculty are infact advisors of the students. Therefore, as the student details are fetched, it is parsed to see if he authored a paper. If he did, his advisor is displayed as the co-author. This optimization results in the avoidance of retrieval of data pertaining to publications and its co-authors.

Query4. Return the faculty who take courses which are attended by the students they advise.

The query is executed by the existing approach by first fetching the details of all the faculty followed by the details of courses taken by them. This is followed by fetching the details of all the students who take these courses. Then the students details are checked to see whether they are guided by the professor who teaches that course. The proposed approach discovers the fact that lecturers take courses but do not advise students. This fact is used to optimize the query by not fetching the details of lecturers's courses and the students who take those courses.

5. CONCLUSIONS AND FUTURE WORK

In this paper we present an approach to optimize the SPARQL query execution over the web of linked data. The approach is a two-phased strategy to discover patterns in the query. These patterns are used to identify data that do not contribute to answer the query and can be prevented from being fetched, resulting in a reduction in the query execution time. We ran simulations that demonstrated the efficacy of the proposed approach. Future work includes the application of proposed approach to optimize conventional SPARQL queries over RDF data.

Table 1: Results

Query	N_e	R_e	N_p	R_p	α	β
Q1	135631	13911	42463	13076	69%	6%
Q2	71714	532	21362	532	71%	0%
Q3	145788	14316	33321	13170	77%	8%
Q4	72044	446	29437	446	59%	0%

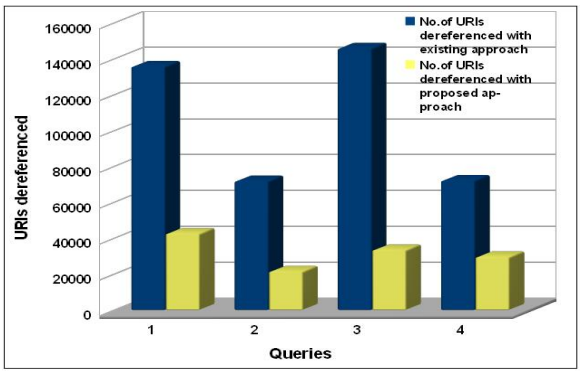

Figure 4: Degree of Optimization achieved

6. REFERENCES

[1] C. Bizer, T. Heath, and T. Berners-Lee. Linked data – the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[2] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, 2007.

[3] M. Franklin. From databases to dataspaces: A new abstraction for information management. *SIGMOD Record*, 34:27–33, 2005.

[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] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 411–420, New York, NY, USA, 2010. ACM.

[6] O. Hartig, C. Bizer, and J.-C. Freytag. Executing sparql queries over the web of linked data. In *8th International Semantic Web Conference (ISWC2009)*, October 2009.

[7] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, pages 774–783. IEEE, 2008.

[8] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, World Wide Web Consortium, 2008.