Hypergraph Based Query Optimization

Sangeeta Sen
BCETW, DURGAPUR
sangeetaaec@ gmail.com

Moumita Ghosh
BCETW, DURGAPUR
moumita4989@ gmail.com

Animesh Dutta
NIT DURGAPUR
animesh.dutta@ it.nitdgp.ac.in

Biswanath Dutta
ISI Bangalore
bisu@drtc isibang.ac.in

Abstract— **Resource Description Framework (RDF) is the fundamental data model for storing information on the web. In recent years, the amount of RDF data available on the web is increasing rapidly. As RDF is a graph based data model, the evaluation of SPARQL queries become expensive. Query optimization and data management problems are the key challenges associated with RDF data. This paper is focused on the concept of hypergraph for storing RDF data and graph based index for query processing. We propose some algorithms for transformation of RDF graph to hypergraph, formation of index, generating query path and for query processing. A PredicateBasedIndex (PBI) is created, from RDF graph, with the help of the size of hyper-edges. According to this index a query path is built for a SPARQL query and executed on the hypergraph. These algorithms are analyzed and compared with some existing literature based on query optimization.**

I. INTRODUCTION

Resource Description Framework1 (RDF) is a core data model for storing information on the Semantic Web [1]. RDF data is stored as triplets (subject or resource, predicate or property and object or property value). The development of" web of data" addresses the problem of data management and query optimization. SPARQL2 is a graph based query language. The main task of SPARQL is to match the graph pattern, which consists of subject, predicate and object, in the underlying RDF graph. In recent years SPARQL is translated into the set of relational queries against the relational DBMS of RDF data [2]. We state that this approach is inefficient for handling SPARQL because it requires as many join operations as in the match pattern of the query. The other important issue is data indexing. Paper [9] discusses three types of indexing technique i) Relational based ii) Entity based and iii) Graph based indexing. Graph index is also used in the paper [7].

In this work, we propose a hypergraph [6], [13] based query optimization and data management technique that has less complexity than the existing techniques discussed in [11], [4], [14], [12]. A hypergraph is a generalization of the graph where the edges connect more than two vertices and such edges are called hyper-edges. We also develop a graph based index for query processing. Following are the main contributions of this paper:

- Predicate based graph index for RDF data: This paper presents a PredicateBasedIndex(PBI) that comprises only the predicates of a RDF graph. The size of this index can be controlled by the number of predicates.
- Query path based query Processing: To process a query, first the model will generate query path based on PBI. According to this path, query is processed in the hypergraph.
- Reduce the joining and union operation cost of relational database.
- Hypergraph based database of RDF data: All subjects and objects connected with a particular predicate resides under a particular hyper-edge, so deletion and insertion of RDF triples become easy to execute.

Rest of the paper is organized as follows. In section II, current state of the art is highlighted. Section III provides an overview of the problem related to SPARQL query language and section IV describes the methodology used to overcome this problem respectively. In section V we give formal definition of the terms used throughout the paper. Section VI demonstrates the proposed algorithms. Explanation of the model with example is shown in section VII. Analysis and comparison of the proposed algorithm with existing algorithm is done in section VIII. In Section IX, lemma proof and metric definitions are shown to verify our proposal. Section X concludes the work mentioning our future work.

II. STATE-OF-THE-ART

There has been a lot of research work in the field of RDF data storage and SPARQL query optimization. The authors of paper [2] describe a novel storage and query mechanism for RDF which works on top of the existing relational database. First contribution is an innovative relational storage representation for RDF data that is both flexible and scalable. Next contribution is SPARQL query optimization and translation of SPARQL to SQL. The paper [6] proposes a hypergraph based storage policy for RDF version management. This scheme uses hypergraph structure in which a hyper-edge contains vertices corresponding to an RDF triple. This enables to reduce the storage space and reconstruct the version with efficiency. Paper [13] proposes statement hypergraph, labeled hypergraph, and query hypergraph. The author provides some advantages over representation of SPARQL as hypergraph such as they can represent RDF data. Authors of the paper [14] introduce "Trinity. RDF", a distributed, memory based graph engine for web scale data. The paper also introduces a new cost model, novel cardinality estimation technique and optimization algorithm for distributed query plan generation. The model of the paper [8] represents an efficient RDF query engine to evaluate SPARQL query. This paper employs inverted index structure for indexing the RDF triple. A tree shaped optimized algorithm is developed that transforms a SPARQL query graph into optimal query plan by effectively reducing the searching. space. A new join ordering algorithm that performs a SPARQL

1www.w3.org/RDF/
2www.w3.org/2001/sw/wiki/SPARQL

978-1-4799-6805-3/15/$31.00 ©2015 IEEE

tailored query simplification is introduced by paper [4]. This paper also represents a novel RDF statistical synopsis that accurately estimates cardinality targeting SPARQL queries. Some challenges in finding optimal join order of SPARQL are i) RDF triple format ii) lack of schema. Dynamic programming based heuristic decomposes the SPARQL graph into disjoint star shaped sub-queries. Most current SPARQL implementations translate a query into a set of relational queries against the underlying relational DBMS. It requires as many joins as the query has triple patterns. The paper [5] provides idea about index, eligible index selection, overlapping index. The paper describes the SPARQL query graph model which is used during query processing and to store additional information about compilation process. It provides a framework for pattern based indexing of RDF database for SPARQL queries. The paper [11] proposes a parameterized structure index called PIG. It is not only used for data partitioning but also designed to calculate the matching of graph structured queries. This paper captures the concept of equal structural neighborhood, known as bi-simulation. The authors of paper [12] focus on providing a data/index structure that supports the efficient solution for queries. The paper provides an algorithm to answer graphical queries by using the GRIN data structure.

In the current work, we propose a hypergraph based query optimization technique in which we want to achieve optimization of user query. From above works, it is evident that translation from SPARQL to SQL takes extra time and computation cost is high. SQL joining and union takes extra computation cost. In paper [8], the query contains predicates as subject in triple patterns. Normally it is not easy to execute the query in a traditional RDF graph because predicate represents a link between subject and object. But since hypergraph subjects, predicates and objects are considered as node so we can use predicates as subjects or objects.

III. PROBLEM SCENARIO

Resource Description Framework (RDF) is a specification developed in 2000 by the W3C as a foundation for processing meta-data regarding resources on the Internet, including the World-Wide Web. SPARQL, a query language, is developed to provide a solution to the problem of RDF data querying. In 2008 SPARQL becomes a W3C recommendation, but query optimization is a big challenge in the domain of semantic web.

To explain the problems we take an example.

RDF graph as shown in Figure. 1, we want to extract information about players, region, club from this graph.

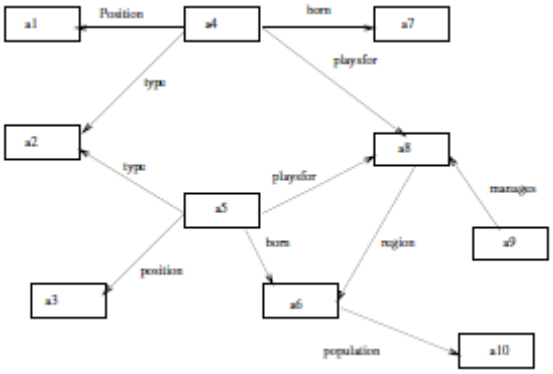


Figure.1. RDF Graph

To extract the above information on players, region, club, we apply SPARQL query1 as shown below. A SPARQL query pattern composed of triples in the form of P(S, O) where S and O are variables. Result of SPARQL query is list of values. It can replace variables in query pattern.

Query1:

```
SELECT ?PLAYER, ?CLUB, ?REGION
WHERE { ?PLAYER playsfor ?CLUB.
?PLAYER type a2.
?PLAYER born ?REGION.
?CLUB region ?REGION.
? REGION population ?POP. }
```

We can represent this query as a graph shown in Figure. 2. The aim, to match this query graph with RDF graph and to get the result with optimized time, is the key challenge.

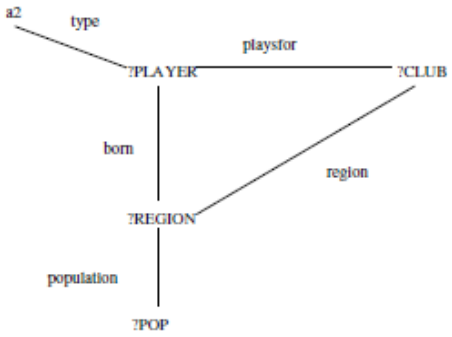


Figure.2. Query Graph for SPARQL query1

Now suppose another SPARQL query to extract the information about player's birth place. The query is

Query2:

```
SELECT ?PLAYER,?REGION
WHERE {?PLAYER born ?REGION.}
```

The query graph for this query is shown in Figure. 3.

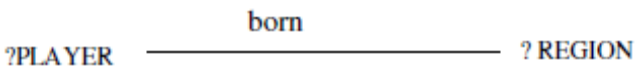


Figure.3. Query Graph for SPARQL query2

Although there is a single predicate match pattern, to match this graph pattern over the RDF graph becomes challenging if the graph is too large. To reduce the query optimization time we use the idea of hypergraph. In succeeding section we discuss our approach in details.

IV. METHODOLOGY

As stated above, in the current work we use the idea of hypergraph, more specifically the undirected hypergraph for optimizing the query process. Before describing the methodology, we briefly discuss the basics of hypergraph.
Hypergraph are of two types:

i) Directed hypergraph [3] or Bipartite hypergraph is a pair of disjoint sets of nodes in which every node is associated with a weight (s,p,o).
ii) Undirected hypergraph [3], [10] is a pair of a set of vertices and set of edges which belongs to the power set of vertices.

Figure. 5 and Figure. 6 represent directed and undirected hypergraph respectively converted from a RDF graph of Figure. 4. Here e1 in directed and undirected hypergraph represents hyperedge.

person1 is Named Tom

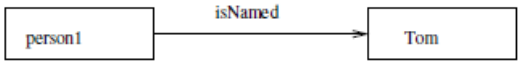


Fig. 4. RDF graph of the example

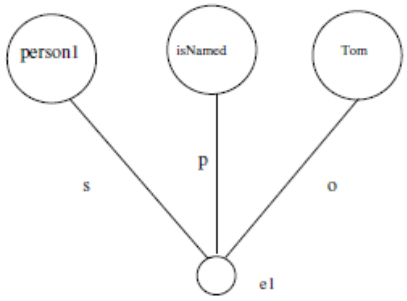


Fig. 5. Directed hypergraph

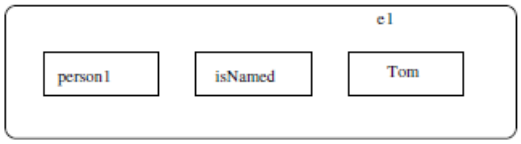


Fig. 6. Un-Directed hypergraph

The proposed model is built using the following steps.

Steps 1: First we convert RDF graph into a hypergraph by calculating the subjects and objects connected with the distinct predicate. To optimize the query processing time, we use undirected hypergraph concept because in a directed hypergraph the hyper-edge is basically a blank node which connects all three subject, object and predicate node with 3 labeled (S, O, P) edges. So each time it creates extra overhead. In order to convert RDF to undirected hypergraph we deploy algorithm 1.

Steps 2: After converting RDF graph to hypergraph the model checks overlapping of hyper-edges to identify the relationship between hyper-edges i.e. the involvement of subjects, objects, predicates of a particular hyper-edge with subjects, objects, predicates of other hyper-edges. Then after calculating the size of each hyper-edge present in the un-directed hypergraph, we sort the hyper-edge and build the index for further calculation. For indexing purpose we use graph based index. Here each predicate forms the node as per size of that predicate's hyper-edge. The predicates are connected with the root, whose hyper-edges size is minimum. Then as per size and overlapping of hyper-edges, an index is created. In order to build the index we utilize algorithm 2.
The following Figure. 7 shows the step by step procedure for building index from RDF graph.

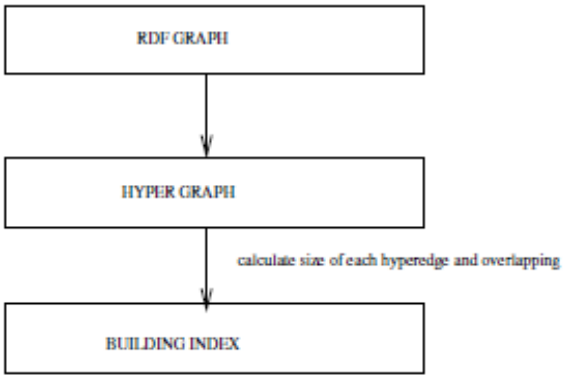


Fig. 7. Transformation of RDF Graph to Index

Steps 3: When a SPARQL query is invoked into the system, we rearrange it according to the predicates' hyperedge size.

Steps 4: Using the modified SPARQL query, we build a query graph. The process of making query graph is explained in algorithm 3

Steps 5: Then based on the index and query graph, query path is created. We use algorithm 4 to represent the creation of query path. Query path describes way in which a query is executed in hypergraph.

Steps 6: According to this path we invoke SPARQL query into the hypergraph i.e. we process the query. To process SPARQL query into undirected hypergraph we build the algorithm 5.
The above mentioned steps (3-6) are shown in Figure. 8.

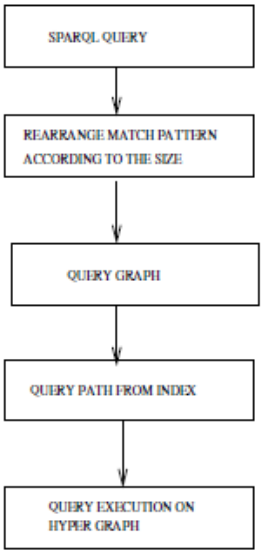


Figure. 8. SPARQL query execution

By using this technique, the advantage is that joining cost of tables becomes less compared to the relational database. Here all the subjects and objects connected with a particular predicate are grouped together under a hyper-edge. So insertion and deletion of data become easy.

V. FORMAL DEFINITIONS

In this section we provide formal definition of the basic terms used in this paper.

Definition 1: RDF graph $G = (V, E)$ where $V = \{v | v \in S \cup O\}$ *and* $E = \{ e_1, e_2..\} \in e = \{u, v\}$ *where* $u, v \in V$. There are two function l_e and l_v. l_e is edge-labeling function. $l_e(S, O) = P$ and l_v is the node labeling function. $l_v(v_t) = t$ where $t \in (S \cup O)$ and S = *Subject (URI* $\cup$ *BLANKS), P=Predicate (URI),* O = *Object(URI* $\cup$ *BLANKS* $\cup$ *LIT).*

Definition 2: Hypergraph $H(G) = (V, E)$ where node $V = \{ v_1, v_2.., v_n \}$ and $E = \{ e_1, e_2,..., e_n \}$ *where* $V = \{v | v \in S \cup O \cup P\}$ and each edge E is a non-empty set of V. $E = \bigcup_{i=1}^{n} V$. If a hypergraph $H(G)$ contains $h_1(G)$, $h_2(G)...h_n(G)$ hyper-edges. So we can say $H(G) = \sum_{i=1}^{n} h_i(G)$.

Definition 3: Undirected Hypergraph $H(G) = (V, E)$ *where* $V = \{ v | v \in S \cup O \cup P\}$ and $E = \sum_{i=1}^{n} e_i$ where e_i is a set of hyper-edges. $\forall P, \forall e \mid (S_i, O_i) \in H(G)$ where $1 \leq i \leq n$. Here S_i, O_i set of subjects and objects of particular predicate P and hyper-edge is e.

Definition 4: Overlapping Hyper-edge $(h_i(G) \sqsubseteq h_{i+1}(G))$ If $h_1(G) = (S_1, P_1, O_1)$ *and* $h_2(G) = (S_2, P_2, O_2)$ *we can say* $h_1(G) \sqsubseteq h_2(G)$ *iff* $\forall s_1 \in S_1 \in h_1(G) \cup s_2 \in S_2 \in h_2(G) \vee \cup o_1 \in O_1 \in h_1(G) \cup o_2 \in O_2 \in h_2(G) \in \in p_1 \in P_1 \in h_1(G) \cup p_2 \in P_2 \in h_2(G)$.

Definition 5: PredicateBasedIndex (PBI) $I(G) = (V, E)$ where $V = \{v | v \cup P_i \in h_i \in \delta \}$ and $E = (v_i, v_j)$ where $v_i, v_j \in V$ and $1 \leq I \leq n-1$, $1 \leq j \leq n$. for $\delta \in V \in e = (\delta, v)$. Here δ represents root of the index.

Definition 6: A SPARQL query (Q^R) mainly contains $< Q^q, Q^s, Q^p >$ where Q^q is the query form and Q^p is the match pattern if $? x \cup var(Q^q)$ then $? x \cup var(Q^p)$ and Q^s contains constraints like FILTER, OPTIONAL. From the SPARQL query we first create query graph (Q^G).

Definition 7: Query graph $(Q^G) = (V, E)$, $V \leftarrow \{ var \cup Q_i^p \}$ and $E \leftarrow \{ P \cup Q_i^p \wedge (var \cup Q_i^p, var \cup Q_{i+1}^p)\}$ where $1 \leq i \leq n$, n is the number of predicates, P is the predicate.

Definition 8: Query path $(Q^{path}) \in Q^{path}, \delta \rightarrow P_i \in I(G) \mid P_i . size$ = min size $\wedge (P_i \rightarrow P_{i+1}) \cup I(G)$ *if* $\forall var \cup Q_i^p = = var \cup Q_{i+1}^p$.

Definition 9: Data Insertion: if query is Q^R and match pattern Q^p then check $\forall P_i \in Q^p \in I(G)$ if true then check $P_i \in H(G)$ $\vee$ create $h_i \in P_i \wedge$ update $H(G)$ with h_i. Then check if $\forall var \in Q^p \in H(G)$ if true then overlap h_i with *var's h* $\in$ update h_i with $var \in Q^p$.

Definition 10: Data Deletion: if query is Q^R and match pattern Q^p then check $\forall P_i \in Q^p \in I(G) \wedge P_i \in$ var if true then check $| h_i | = = 0$ if true then remove $var \cup Q^p \in H(G) \vee$ copy *of* P_i exists.

Following are the algorithms to accomplish the steps mentioned in section IV.

VI. PROPOSED ALGORITHMS

Before discussing the algorithms, we define the symbols used across the algorithms.

↓ is used to represent successor i.e. x↓y gives the meaning y is the successor of x, ↔ is represents that left and right variables connected with ↔ symbol are associated with each other i.e. x ↔y means x and y associated with each other, ⟶ provides single direction connection i.e. if a⟶ b then we can understand that direction of the connection is from a to b, δ represents the root.

In proposed hypergraph each hyper-edge is created for each predicate present in the RDF graph and its related subjects and objects as shown in algorithm 1.

Algorithm 1: **Algorithm to create Hypergraph**

INPUT: RDF graph G as triple pattern *(S, O, P)*

OUTPUT: Hypergraph Graph *(V, E),* where *V* is set of subjects, objects and predicates and E is edges covering subjects, objects of particular predicate.

1: $V \leftarrow \phi$, $E \leftarrow \phi$, $i \leftarrow 0$, $j \leftarrow 0$, $e_i \leftarrow \phi$
2: **for all** *(S, O, P) ∈ G* do
3: $V \leftarrow V \cup V \in (S \cup O \cup P)$
4: **end for**
5: **for all** *Pi* where *1< i <n do*
6: **for** *1< j < n do*
7: $e_i \leftarrow \{$ Pi, {*Sj, Oj* } }
8: j ←*j+1*
9: **end for**
10: *Ei←ei*
11: *i←i+1*
12: **end for**
13: **Return** *H(G) = (V, E*)

end

To build the index we use predicates present in hypergraph as per hyper-edges' size and the overlapping between hyper-edges as shown in algorithm 2.

Algorithm 2: Algorithm to create PredicateBasedIndex(PBI)

INPUT: Hypergraph *H(G).* If a hypergraph *H(G)* contains $h_i(G), h_2(G)...h_n(G)$ hyper-edges

OUTPUT: PredicateBasedIndex (*PBI) I(G)*

1: j← 1
2: Sort hyper-edges as per size
3: **for all** h_i where *1< i < n-1* **do**
4: **if** (*MIN*(h_i *.size())*) **then**
5: *I(G)=I(G)* ∪ $S \downarrow P_i$
6: **end if**
7: **for** *i+1<j<n* **do**
8: **if** $(h_i(G) \sqsubseteq h_j(G))$ *AND* $(P_i.\ size == P_j.\ size)$ **then**
9: *I(G)=I(G)* ∪ $P_i \leftrightarrow P_j$
10: **else**
11: *I(G)=I(G)* ∪ P_i —> P_j
12: **end if**
13: *j← j+1*
14: **end for**
15: *i ← i+1*
16: **end for**
17: **Return I(G)**.

From the SPARQL query a query graph is built with the help of the match patterns as shown in algorithm 3.

Algorithm 3: **Algorithm to create Query graph**

INPUT: *Query match pattern* $Q^P \cup Q^R$

OUTPUT: *Query graph* Q^G

1: $V \leftarrow \phi$, $E \leftarrow \phi$, *i←1, j←1*
2: **for all** Q_i^P *where 1< i* **do**
3: **if** *i = = 1* then
4: $V \leftarrow V \cup$ *(var* ∪ Q_i^P*. S* ∪ *var* ∪ Q_i^P*. O)*
5: $E \leftarrow$ *(var* ∪ Q_i^P*. S, var* ∪ Q_i^P*. O)*
6: **end if**
7: **if** *i >2* **then**
8: **if** *(var* $\in Q_i^P$*. S = = var* $\in Q^P{}_{i+1}$*. S)* AND *(var* $\in Q_i^P$*. O = = var* $\in Q^P{}_{i+1}$*. S)* **then**
9: $V \leftarrow V \cup$ *var* ∪ Q_i^P*. S*
10: $E \leftarrow$ *(var* ∪ Q_i^P*, var* ∪ $Q^P{}_{i+1}$*)*
11: **else**
12: $V \leftarrow V \cup$ *var* ∪ $Q^P{}_{i+1}$*.O*
13: $E \leftarrow$ *(var* ∪ Q_i^P*, var* ∪ $Q^P{}_{i+1}$*)*
14: ***end if***
15: **else**
16: **if** *(var* ∪ Q_i^P*. S = = var* ∪ $Q^P{}_{i+1}$*. O) AND (var* ∪ Q_i^P*.O = = var* ∪ $Q^P{}_{i+1}$*. O)* **then**
17: **if** ∪ var ∪ $Q^P{}_{i+1}$*.* S ∪ V **then**
18: $V \leftarrow V \cup$ *var* ∪ Q_i^P*.S*
19: $E \leftarrow$ *(var* ∪ Q_i^P*, var* ∪ $Q^P{}_{i+1}$*)*
20: **end if**
21: **else**
22: $V \leftarrow V \cup$ *var* ∪ $Q^P{}_{i+1}$*.S*
23: $E \leftarrow$ *(var* ∪ Q_i^P*, var* ∪ $Q^P{}_{i+1}$*)*
24: **end if**
25: **end if**
26: *i* ← *i+1*
27: **end if**
28: **Return** $Q^G = (V, E)$

After creating query graph, we build query path from index shown in algorithm 4.

Algorithm 4: Algorithm to create Query path

INPUT: Sets of predicates $P \cup Q^G$, Q^G *and I(G)*

OUTPUT: Query graph Q^{Path}

1: Start from *δ, f←ϕ, i←1*
2: $Q^{Path} \leftarrow \delta \rightarrow min\ (P_i.\ size) \cup I(G)$
3: **for** *i ←1 to n-1* **do**
4: **for** j← i+1 to n **do**
5: **if** ∪ var ∪ Q_i^P ∪ Q^G = = var∪ Q_i^P ∪ Q^G **then**
6: $Q^{Path} \leftarrow Q^{Path} \cup (P_i \rightarrow P_j) \cup I(G)$
7: f ←*1*
8: **if** *(f = =1)* **then**
9: **break**
10: **else**
11: *j←j+1*
12: continue
13: **end if**
14: **end if**
15: *i←i+1*
16: **end for**
17: **end for**
18: **Return** Q^{path}

End

To execute the query, shown in algorithm 5, from the query path and hypergraph we use two functions

i) *loadnode(match pattern(access variable)):* This function returns access variables i.e. subject or object or both based on the predicate present in Match pattern is described in algorithm 6;

ii) *loadneighbournode(match pattern, match pattern(access variable)):* This function returns access variables i.e. subject

or object or both after comparing two match pattern's subject position variable and object position variable is shown in algorithm 7.

Algorithm 5: Algorithm to create Query processing

INPUT: Q^G, Q^P and *I(G)*
OUTPUT: Query Answer Q^{AG}

1: **for all do**
2: Q_i^P where i 1 to n-1
3: **if** *i = = 1* **then**
4: *loadnode*(Q_i^P (S, O))
5: **end if**
6: **if** i > 2 **then**
7: *loadneighbournode* (Q_i^P, $Q^P{}_{i+1}$(S, O))
6: **end if**
8: $i \leftarrow i+1$
9: **end for**
10: **Return** Q^{AG}
end

Algorithm 6: ***Algorithm for loadnode*** **(Q_i^P (S, O))**

1: $V \leftarrow \phi$, $E \leftarrow \phi$, $i \leftarrow 1$, $j \leftarrow 1$
2: *execute query on* h_i
3: $v_i \leftarrow$ S $\wedge w_j \leftarrow$ O
4: $V \leftarrow \{ vi , wj \}$
5: $E \leftarrow (vi , wj)$
6: $Q^{AG} \leftarrow (V, E)$
7: *Return* Q_i^{AG}

Algorithm 7: ***Algorithm for loadneighbournode*** **(Q_i^P , $Q^P{}_{i+1}$ (S, O))**

1: $V \leftarrow \phi$, $E \leftarrow \phi$
2: **if** ($Q_i^P.S == Q^P{}_{i+1}.S$)*AND* ($Q_i^P.O == Q^P{}_{i+1}.S$) **then**
3: **if** □ $Q_{i+1}{}^P.O$ □ V **then**
4: **V**←V ∪ $Q_i^P.S \wedge Q_i^P.O \in Q^P{}_{i+1}.O$
5: **E**←$(V \in Q_i^P, V \in Q^P{}_{i+1})$
6: **else**
7: $V \leftarrow V \cup Q^P{}_{i+1}.O$
8: **E**←$(V \in Q_i^P, V \in Q^P{}_{i+1})$
9: **end if**
10: **else**
11: **if** ($Q_i^P.S == Q^P{}_{i+1}.O$)*AND* ($Q_i^P.O == Q^P{}_{i+1}.O$)**then**
12: **if** □ $Q^P{}_{i+1}.S$ □ V **then**
13: **V**←V ∪ $Q_i^P.S \wedge Q_i^P.O \in Q^P{}_{i+1}.S$
14: **E**←$(V \in Q_i^P, V \in Q^P{}_{i+1})$
15: **end if**
16: **else**
17: $V \leftarrow V \cup Q^P{}_{i+1}.S$
18: **E**←$(V \in Q_i^P, V \in Q^P{}_{i+1})$
19: **end if**
20: $Q^{AG} \leftarrow Q_i^{AG} \cup$ (V,E)
21: **Return** Q^{AG}
end

VII. EXPLANATION W ITH EXAM PLE

Following the model discussed above, we convert the RDF graph shown in Figure. 1 into undirected hypergraph shown in Figure. 9.

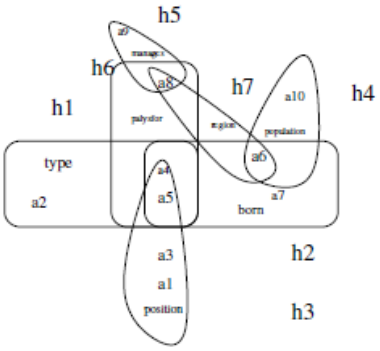


Figure. 9. Hypergraph

After calculating size of the each hyper-edge in hypergraph we find type=2, playsfor=2, born=2, position=2, Manages=1, region=1, population=1. Following this we build the index shown in Figure. 10 where minimum size hyper-edges are con- nected with root.

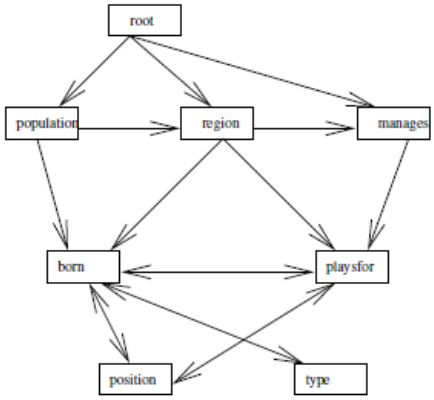


Figure. 10. Index file

For SPARQL query1 (see section III), after rearranging according to predicates' hyper-edge size, the query becomes.

SELECT ? PLAYER, ? CLUB, ? REGION
WHERE { ? REGION population ? POP.
? CLUB region ? REGION.
? PLAYER playsfor ? CLUB.
? PLAYER born ? REGION.
? PLAYER type a2.}

From this query we first build the query graph as shown in Figure. 11.

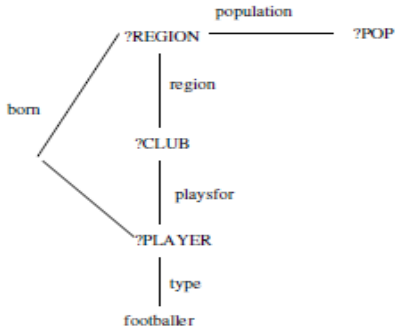


Figure. 11. Query graph

From this query graph and index we build the query path shown in Figure. 12.

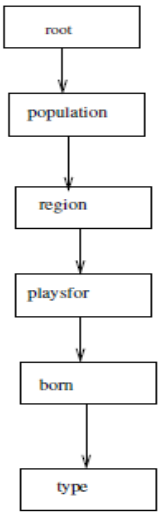


Figure. 12. Query path

According to the above query path we process the query described in Figure. 13.

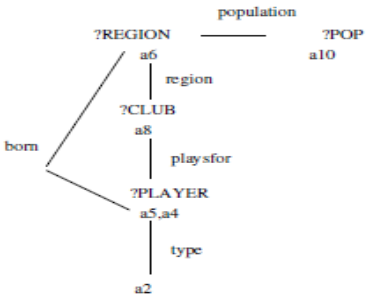
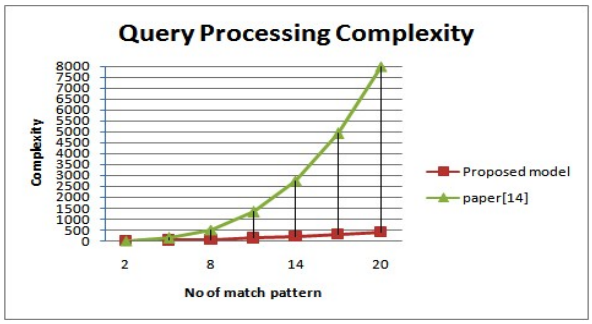


Figure. 13. Query Processing

After the query processing we get the result ?PLAYER=a5, ?CLUB=a8, ?REGlON=a6. Here we do not get a4 because a4 is not connected with born predicate with predefined region a6.The second problem also can be solved by above algorithms. Due to space limitation, we avoid to illustrating the solution

VIII. ALGORITHM ANALYSIS AND COMPARISON

The authors of the paper [11] provide a algorithm for evaluating Match classes. If the first loop *p(x, y), x—»*s and y—>*o,* iterates *n* times then complexity for this algorithm is $O(n^3)$. In the paper [4], they provide a general SPARQL join ordering algorithm. This algorithm calls three more algorithms for getting the result. The maximum complexity is $O(n^3)$ where *n* is the number of triple pattern. The paper [12] provides complexity for building the index is O(|R|4 *log2* |R|)and GRINAnswer algorithm have complexity *O*(l*R*l! *)* where R represents resource of the RDF graph. The complexity of the algorithm (MatchPattern(e)) of paper[14] is also *O(n3)* where *n* is the number of match pattern. The algorithm is also divided into two parts. The maximum complexity among the two parts is *O(n3).* Our proposed model has the complexity of building index is $O(n^2)$ where *n* is the number of hyper-edges for particular predicate and the complexity of the query processing is *O(n)* where *n* is the number of match pattern in the query.



On comparing our query processing algorithm with the algorithm (MatchPattern(e)) of paper[14], we get the graph which is shown in Figure. 14.

IX. LEMMA AND METRIC DEFINITION

Lemma 1: Two hyper-edges can be used to form a third hyper-edge containing elements of both hyper-edges iff they have common element.

proof: Let H(G) be the hypergraph of a set of hyper-edges h. For any two hyper-edge $h_1(G)$, $h_2(G)$ □ H(G). If $h_1(G)=(S_1, P_1, O_1)$ and $h_2(G) = (S_2 , P_2 , O_2)$ □ □ s_1 □ S_1 □ $h_1(G)$ □ s_2 □ S_2 □ $h_2(G)$ □ □ o_2 □ O_2 □ $h_2(G)$ □ □ p_2 □ P_2 □ $h_2(G)$ □ □ o_1 □ O_1 □ $h_1(G)$ □ o_2 □ O_2 □ $h_2(G)$ □ □ s_2 □ S2 □ $h_2(G)$ □ □ p_2 □ P_2 □ $h_2(G)$ □ □ p_1 □ P_1 □ $h_1(G)$ □ p_2 □ P_2 □ $h_2(G)$ □ o_2 □ O_2 □ h2 (G) □ □ s_2 □ S_2 □ h_2 (G). Since we can say h_1 (G) h_2 (G), we can create a third hyper-edge with (s_1 or s_2, p_1 or p_2, o1 or o_2)

Lemma 2: Size of the resultant hyper-edge will be less than or equal to the smaller of the two hyper-edges used to form it.

proof: For any two hyper-edge h1(G), h2(G) □ H(G). Let |h1(G)|= n □ |h2(G)| = k □ resultant hyper-edge h3(G)| = x □ n>k, now we try to find out common variables between these two hyper-edge by h1(G) ∩ h2(G). Now if a third hyper-edge h3(G) is formed from the variables of the previous operation and its size is x then

case 1: All variables of $h_2(G)$ are also common to $h_1(G)$. Then resultant hyper-edge $h_3(G)$ will be formed with k number of variables thus $|h_3(G)|=x=k$.

case 2 : Some variables of $h_2(G)$ i.e. x such that x<k are common to $h_1(G)$. Then the size of $|h_3(G)|$ will be x<k

Thus in no circumstances all the elements of $h_1(G)$ can be common to those $h_2(G)$. Since $h_2(G)$ contains less elements than $h_1(G)$. Therefore from the above two cases we can say that $h_3(G)$ will have elements either equal to or less than what $h_2(G)$ has or $|h_3(G)|=x \leq k$ i.e. $|h_3(G)| \leq |h_2(G)|$.

Lemma 3: Selection of query path directly proportional to the size of predicates' hyper-edge and inversely proportional to the PredicateBasedIndex I(G). The formula for query selection is

$$\frac{\prod_{i=1}^{n} P_i \in |h_i(G)| \in Q^p}{I(G)^{mintomax|h_i(G)| \wedge h_i(G) \sqsubseteq h_{i+1}(G)}}$$

Proof: Hyper-edge's predicate connected with root suppose $h_1(G)$ and $h_2(G)$ have the lowest size. $h_3(G)$ and $h_4(G)$ have the same size but greater than $h_1(G)$ and $h_2(G)$. Now suppose $h_1(G)$ $h_3(G)$ $h_4(G)$ and $h_2(G)$ $h_4(G)$ □ P_1□$h_1(G)$, P_2 □ $h_2(G)$, …. P_n □ $h_n(G)$ then if a match pattern contains P_1, P_3,and P_4 predicates then

Selection of query path is

$$\frac{P_1 \in h_1(G), P_3 \in h_3(G), P_4 \in h_4(G)}{I(G)^{h_1(G), h_2(G), h_3(G), h_4(G)}} \text{ or}$$

$$\frac{P_1 \in h_1(G), P_4 \in h_4(G), P_3 \in h_3(G)}{I(G)^{h_1(G), h_2(G), h_3(G), h_4(G)}}$$

Metric definition

i) Total computation cost of making hypergraph (H(G)) based on RDF graph(G):Identify the Headings

$$\text{cost(H(G))} = \sum_{i=1}^{n} cost(V_i.E_i) \text{ Where } V_i \text{ and } E_i \in G$$

If the RDF graph has N vertices and M edges then the computation cost of producing hypergraph becomes computation cost of (M*N) = x.

Case 1: If vertices become N+1 and edges become M+1 then the computation cost is (N+1)(M+1) i.e. (N+M+NM+1)=y>x

Case 2: If vertices are N and edges are M+1 then the computation cost is N*(M+1) i.e NM+N=z>x

So we can conclude that smaller the RDF graph lesser the computation cost.

ii) Total computation cost of query processing (Q^{pro}) based on Q^{path}:

$$cost(Q^{pro}) = \sum_{P_i \in Q^{path}} cost(P_i)$$

If the predicates in match pattern (in query path also) is n then the computation cost of query processing = computation cost of n predicates. Now if predicates become n+1, we can say that computation cost of n+1 predicates > computation cost of n predicates.

So if the predicates of query path are less then the computation cost of query processing is also less.

X. CONCLUSION

In this paper we provide a framework for hypergraph and graph based index for RDF data. We focus on query processing by creating query path from graph based index (Predicate Based Index) and execute it into the hypergraph. After comparing the complexity of our algorithm with other existing algorithms, we state that our proposed algorithms are less complex than others.

Future works include the implementations of our approach, the evaluation of graph based indexing for query processing and clustering of hypergraph if the graph is big enough. In the present work, we only considered the simple queries. Our approach can be further extended with more complex queries.

REFERENCES

[1] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. 2001

[2] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient rdf store over a relational database. pages 121–132, 2013.

[3] Frithjof Dau. Rdf as graph-based, diagrammatic logic. pages 332–337,2006.

[4] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in sparql queries. In EDBT'14, pages 439–450, 2014.

[5] Ralf Heese, Leser, Ulf, Quilitz, Bastian, Rothe, and Christian. Index support for sparql. In ESWC, 2007.

[6] Dong-Hyuk Im, Nansu Zong, Eung-Hee Kim, Seokchan Yun, and Hong-Gee Kim. A hypergraph-based storage policy for rdf version management system. 2012.

[7] Kisung Kim, Bongki Moon, and Hyoung-Joo Kim. Rg-index: An rdf graph index for efficient sparql query processing. 2014.

[8] Chang Liu, Haofen Wang, Yong Yu, and Linhao Xu. Towards efficient sparql query processing on rdf data, 2010.

[9] Yongming Luo, Francois Picalausa, George H.L. Fletcher, Jan Hidders, and Stijn Vansummeren. Storing and indexing massive rdf datasets. In Semantic Search over the Web, pages pp 31–60, 2012.

[10] Amadis Antonio Martinez Morales. A directed hypergraph model for rdf. In Elena Paslaru Bontas Simperl, Jrg Diederich, and Guus Schreiber, editors, KWEPSY, volume 275 of CEUR Workshop Proceedings. CEUR-WS.org, 2007.

[11] Gunter Ladwig Thanh Tran. Structure index for rdf data. 2010.

[12] Udrea, Octavian, Pugliese, Andrea, Subrahmanian, and V. S. Grin: A graph based rdf index. In AAAI, pages 1465–1470. AAAI Press, 2007.

[13] Pingpeng Yuan, Wenya Zhang, Hai Jin, and Buwen Wu. Statement hypergraph as partitioning model for rdf data processing. pages 138–145, 2012

[14] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. pages 265–276, 2013.