Designing Indexing Structure for Discovering Relationships in RDF Graphs

Stanislav Bartoň

Faculty of Informatics, Masaryk University, Brno, Czech Republic
`xbarton@fi.muni.cz`

Abstract. Discovering the complex relationships between entities is one way of benefitting from the Semantic Web. This paper discusses new approaches to implementing ρ-operators into RDF querying engines which will enable discovering such relationships viable. The cornerstone of such implementation is creating an index which describes the original RDF graph. The index is created in two steps. Firstly, it transforms the RDF graph into forest of trees and then to each tree creates its extended signature. The signatures are accompanied by the additional information about transformed problematic nodes breaking the tree structure.

1 Introduction

One form of retrieving information from the Semantic Web is to search for relations among entities. The simple relations such are the *is-a* or *is-part-of* relations can be found easily. For example using RQL [3] one can find direct relationship among entities. This means that we are able to retrieve all the descending classes of one class, even on a different level. For example the user can ask for all instances of a class 'artist' as it is shown in Figure 1. The answer to such query would be all instances of both its subclasses in the knowledge base, all painters and sculptors. But in the Semantic Web there can be observed more complex relationships among entities [7] than those simple ones.

Such complex relationship can be represented by a path between two entities consisting of other entities and their properties. To discover such complex relationships ρ-operators [1] have been developed. In this paper, the complex relationships are discussed and are referred to as Semantic Associations [7]. The ρ-operators are precisely the tools for discovering such Semantic Associations. This class contains ρ *path*, ρ *connect* and ρ *iso* operators.

ρ **path** - This operator returns all paths between two entities in the graph. An example of such relation can be seen in Figure 1 between resources `&r6` and `&r8`. Such association represents an information that a painter called Pablo Picasso had painted a painting which is exhibited in Reina Sofia Museum.

ρ **connect** - This one returns all intersecting paths, on which the two entities lie. An example of those two intersecting paths is the one between resources `&r6` and `&r8` and between resources `&r9` and `&r8`. This association represents a fact that two artist had their artifacts (in one case it was a painting and in the other a sculpture) exhibited in the same museum.

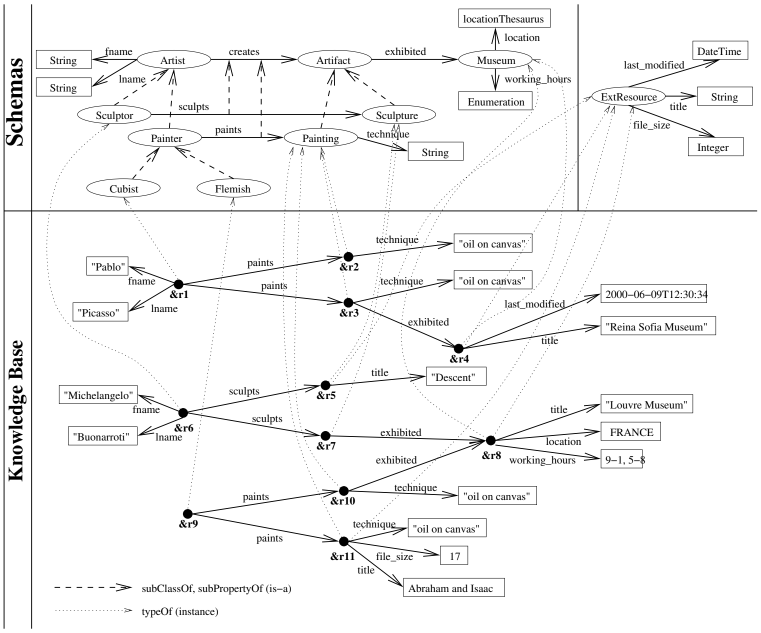

Fig. 1. An example of RDF graph

ρ **iso** - This operator implies a similarity of nodes and edges along the path, and returns such similar paths between entities. An example of such association is the relation between resources `&r1` and `&r9`. This represents a fact that both subjects are classified as painters.

The ρ iso operator should also return an information that the subjects are artists besides that they are both painters. Ranking of such answers is very important since the fact that two subjects are painters is obviously more relevant than they are artists, due to its greater specialization. The relevance of relations found of course significantly depends on the context in which are the queries asked.

The possible usage of searching such complex associations can found in the field of national security. For example the system could find its usage on airports helping to identify suspicious passengers by looking for available connections between them.

In this paper we mainly focus on the former two operators which are the ρ path and ρ connect. We introduce a design of a indexing structure for the RDF graph that will make the discovery of the relationships described by these ρ operators effective.

Section 3 discusses the related work to the topic of indexing RDF graphs. Section 2 contains a brief introduction into the RDF and the RDF Schema. In Section 4 we present out contribution to the issue by introducing the transformation of the RDF graph into forest of trees and after-wards the application of tree signatures to those trees. Section 5 outlines possible improvements to the indexing structure that is designed in this paper. Finally Section 6 concludes the whole paper.

2 Preliminaries

The RDF graph depicted in Figure 1 is visualization of an RDF and RDF Schema notation. These two languages are used to state the meta information about resources. The following subsections briefly describe this technology. In the scope of this paper the RDF is used to create the knowledge base and the RDF schema to build the schema parts of the RDF graph.

2.1 RDF

The abbreviation RDF stands for resource description framework and according to [4] is supposed to be a foundation for processing metadata. It basically provides a data model for describing machine-processable semantics of data. The RDF statement is a triple (O, P, V) which parts stand for object, property and value. Object is usually identified by URI. It is basically a resource. The value can be either an explicit value or a resource also. Since this triple itself can be considered as a resource it can appear in an RDF statement as well. This means that the data model can be envisioned as a labeled hypergraph (each node can be an entire graph) where an edge between two nodes represents the property between the object and value.

2.2 RDF Schema

Because the modeling primitives of RDF are so basic, there is no way to define the class-subclass relation. Therefore an externally specified semantics to some resources was provided. Such enriched RDF is called RDF Schema [2]. Those specific resources are for example rdfs:class and rdfs:subclass.

In such enriched environment we are able to define a simple model of classes and their relations. This can be used to define simple ontologies in the web space. The RDF Schema statements are expressed using XML together with its specific namespace. Even RDF statements can be expressed using XML with its specific namespace.

3 Related work

To make the best of these operators, they should be implemented into an RDF querying system. One of such implementation is presented in [5]. The effort described there demonstrates an implementation of ρ path operator above the RDF Suite [3]. The implementation cornerstones are two indices, the Path index

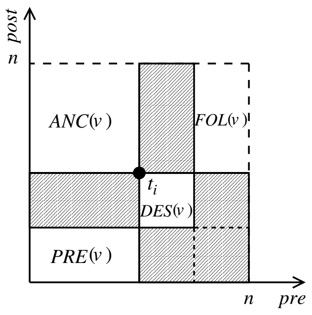

Fig. 2. Properties of the preorder and postorder ranks.

and a Schema index. The former one is two-dimensional array of paths - it carries the information about all paths between Class i and Class j in the schema part of the RDF graph. The latter one is used to search for a path between classes in different schemas. The Path index is very memory intensive when the data grows to large amounts. Therefore this paper discusses a different approach to index the data for the purpose of discovering Semantic Associations.

4 Indexing RDF graphs

The idea of indexing RDF graph demonstrated in this paper is to transform it into tree or forest of trees in which the searching for relationship between particular nodes will be much easier than in general directed graph. If we consider ρ-path and ρ-connect operators, the problem is to find certain paths among particular nodes. Therefore we deploy convenient indexing structure to each tree to optimize such searching. Thus the signature [8] to each tree will be created. This approach solves the problem of getting the relationship between each pair of nodes in a tree by an atomic operation. Such relationship between two nodes in a tree is represented by their mutual position in such tree (i.e. ancestor, descendant, preceding or following). Tree signatures are described in the following subsection.

4.1 Tree signatures

The idea of the tree signature is to maintain a small but sufficient representation of the tree structures. The preorder and postorder ranks[1] are used as suggested in [6] to linearize the tree structure.

The basic tree signature is a list of pairs. Each pair contains a tree node name along with the corresponding postorder rank. The list is ordered according to the preorder rank of nodes.

[1] How the preorder and postorder ranks are obtained please refer to [8].

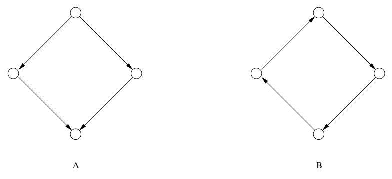

Fig. 3. Directed graphs that are not trees.

Given a node v with particular preorder and postorder ranks, their properties can be summarized in a two-dimensional diagram, as illustrated in Figure 2, where *ancestors ANC(v)*, *descendants DES(v)*, *preceding PRE(v)*, and *following FOL(v)* nodes of v in the particular tree are clearly separated in their proper regions. Due to these properties the mutual position of two nodes within one signature is clear immediately after reading a record of either of them in the particular signature.

According to the signature structure the basic tree signature can be further extended. To each entry a pair of preorder numbers is added. Those numbers represent pointers to the *first following*, and the *first ancestor* nodes of a given node. If no terminal node exists, the value of the first ancestor is zero and the value of the first following node is $n+1$, where n is the number of nodes in a tree. Such enriched signature is called *extended signature*. Later on when we refer to signature we will mean the extended one.

4.2 Transforming the graph into forest of trees

The structure of the RDF Schema and the knowledge base can be envisioned as a directed graph with arcs provided with labels, example is shown in Figure 1. The inconvenience of this structure lies in the problem of searching path between nodes. Such searching algorithms work with great time computational complexity.

Because the structure depicted above is not really a general directed graph, we can get the benefit of the schema part of the structure since it carries useful information about the knowledge base. The schema part has the same function as a schema in the relational database. Then if we could reduce the problem of searching in the whole graph to the problem of searching in the schema, which is considerably smaller, we could use the same algorithms with better time complexity results. But since the graph can contain several schema definitions and the resources can be derived from more than one schema, the desired paths can only be found using the real data, because they would not be included in the schema definition.

Knowledge base transformation A tree can be defined as a directed graph in which is true that (1) each node has zero or one incoming edge and (2) it does not contain a cycle. Directed graphs marked as A and B depicted in Figure 3 break those rules respectively. The transformation of the directed graph into forest of trees lies in the removal of such problematic cases.

If we consider the problem marked as (A) in Figure 3, part (1) in Figure 4 shows a transformation to achieve structure conforming to the rule marked as (1). The black node in the phase 1 in Figure 4, means that the node will be 'divided' into two nodes in the following phase. The next phase has two alternatives, phase 2a demonstrates the division of a node with a duplication of all descendants to all divided nodes. Phase 2b shows the division without duplication. The right way to handle such situation is to use the former method since it prevents the uncontrollable growth of the structure. This assures that the structure will grow in linear space instead of possible exponential growth. The descending nodes should be cut off into stand alone component to avoid 'short cuts' within one component. This becomes important in the moment of finding paths between nodes.

Thus the whole graph is traversed and all the nodes that have more than one incoming edge are divided into exact amount of nodes that is the number of that node's incoming edges. This transformation can lead to breaking the graph[2] into several components. These components are either trees or directed graphs containing a cycle. To identify which components are trees a rule that a graph is a tree only if it has exactly *n+1* edges, where *n* represents the number of nodes in a particular component. The non-tree components are then transformed as follows.

The transformation of the directed graph containing a cycle is depicted in the part marked as (2) in the Figure 4. The spanning tree of such component is found and the nodes, which edges are not contained in the spanning tree are divided. The transformation works in the way that it divides the particular node into two, that the first one contains all the edges that have the original node as the terminal one, and the extra node has all the edges that had the original as a initial one.

[2] We consider that at the beginning the graph consists from only one component.

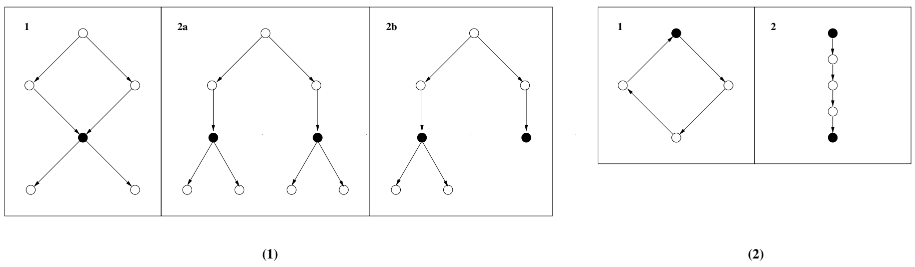

Fig. 4. Transformation of a graph to conform with rules (1) and (2) respectively.

Obviously, after transforming all the non-tree components, we get a forest of trees representing the original graph. Of course we have to store the information about the divided nodes to assure that no information contained in the original graph will not be lost in the new structure. Such information is stored in two tables where the first one is used to get all the multiple nodes[3] in the particular signature, and the second table stores to each multiple node all signatures it appears in. Those two tables connect the components back into the original graph.

The time computation complexity of the transformation of a general directed graph into forest of trees is estimated to $\mathcal{O}(2n)$ in the worst case. It mostly depends on the number of components which after the first transformation contain cycle. The spanning tree to such component has to be built but that means that all the nodes of such components has to be traversed again. If all the newly built components have cycles we have to traverse the whole graph again.

4.3 ρ path and ρ connect implementation

Once we obtain the desired forest of trees we create a signature to each component (tree) of the transformed graph which together with the additional information about multiple nodes will represent the index to the original RDF graph. The time computational complexity of such operation is equal to $\mathcal{O}(n)$ since the algorithm used traverses each node in each component once. The additional information connecting signatures together is built along and deploys only atomic operations. Such information about the multiple nodes is represented by two tables. One has in each row a name of a multiple node together with a particular signature or signatures it appears in. And the other one has a row for each signature with a list of multiple nodes contained in it.

Above such index an algorithms implementing the ρ path and ρ connect operators have been designed. The outline of those algorithms are demonstrated in Algorithm 1 and Algorithm 2.

Algorithm for discovering paths The first algorithm returns an answer whether there exists a path between two nodes. The algorithm traverses the forest of trees only in one direction, so to tell whether there is really a path between two nodes we have to switch the start and end node and deploy the algorithm again if the search has not been successful for the first time. As a by-product it also creates a list of multiple nodes that lie on the path between the two nodes. The exact path is not computed at this point. Another function, to which this list is passed, takes care of the exact path computation. To make the most from the tree structure of this index, the path is computed from the bottom to the top, the first ancestor pointer from the signature is used to traverse the path.

[3] A node which was represented as a one in the original graph, but is represented by several nodes in the new structure.

Algorithm 1 Name: findPathUp

Input: startSignature, startNode, checkedMultiples, endNode, wholePath
Output: true if the path between startNode and endNode exists else false

```
returnValue = false;
if startSignature.isNodeInSig(endNode) then
  return startNode.isDescendantOrSelfOf(endNode);
else
  multiplesInSignature = getMultiplesInSignature(startSignature);
  if multiplesInSignature.isEmpty() then
    return false
  end if
  Set usableMultiples = $\emptyset$;
  for all multipleNode in multiplesInSignature do
    if multipleNode.isAncestorOrSelfOf(startNode) AND !checkedMultiples.contains(multipleNode) then
      usableMultiples.add(multipleNode);
    end if
  end for
  if usableMultiples.isEmpty() then
    return false;
  end if
  for all usableMultiple in usableMultiples do
    usableSigantures = getSignaturesToMultiple(usableMultiple);
    checkedMultiples.add(multipleNode);
    for all usableSignature in usableSignatures do
      uniqueNodes = getSetOfMultipleNodeNamesInSignature();
      for all uniqueNode in uniqueNodes do
        if !checkedMultiples.contains(uniqueNode) then
          if findPathUp(usableSignature, uniqueNode, checkedMultiples, endNode, wholePath) then
            wholePath.add(uniqueNode);
            returnValue=true;
          end if
          checkedMultiples.add(uniqueNode);
        end if
      end for
    end for
  end for
end if
return returnValue;
```

Therefore the algorithm traverses the index structure in only one direction, from bottom to top, it has to be deployed twice unless the path has not been found in the first deployment. Thus to check whether there is not a path between two nodes we have to execute the algorithm twice with both nodes used as a starting point respectively. This implies that the time computational complexity of finding a path between two nodes mainly depends on existence of such path. The problem of dual execution could be solved if we could tell the mutual position of the two nodes in the indexing structure. Then we could deploy the algorithm exactly once to tell whether there exist a path between the two nodes or it does not.

Algorithm for discovering connections As for the ρ connect operator, the nature of the designed index structure implies that the connection, the intersecting node, can only be a multiple node. Therefore the problem of finding two paths that intersect is reduced to finding a multiple node, to which exists a path

Algorithm 2 Name: findConnection

```
Input: node1, node2
Output: node where the two paths intersect or null
Set checkedMultiples1, checkedMultiples2, testedIntersections = $\emptyset$;
List multiples1, multiples2, toDoMultiples1, toDoMultiples2, wholePath1, wholePath2 = $\emptyset$;
done=false; found = false;
while !done do
  Set usableMultiples1, usableMultiples2;
  checkOneUsableMultiple(toDoMultiples1, usableMultiples1, checkedMultiples1, node1, node1Signature);
  checkOneUsableMultiple(toDoMultiples2, usableMultiples2, checkedMultiples2, node2, node2Signature);
  toDoMultiples1.addAll(usableMultiples1); multiples1.addAll(usableMultiples1);
  toDoMultiples2.addAll(usableMultiples2); multiples2.addAll(usableMultiples2);
  testMultiples = (multiples1 $\cap$ multiples2) - testedIntersections;
  if !testMultiples.isEmpty() then
    intersection = testMultiples.get(0);
    intSignature = getSignature2Node(intersection);
    testedItersections.add(intersection);
    if findPathUp(intSignature, intersection, node1, wholePath1) AND findPathUp(intSignature, intersection, node2, wholePath2) then
      return intersection;
    end if
    if toDoMultiples1.isEmpty() AND toDoMultiples2.isEmpty() then
      done=true;
    end if
  end if
end while
return null;
```

from either node. The outlined Algorithm 2 searches the index structure in a direction that the edges have. Its starting nodes are the two nodes to which it is looking for connection.

Throughout the algorithm a set of multiple nodes, nodes which lie below the particular starting node and are possible intersection, a set of checked nodes, nodes through which the algorithm already switched to different signatures and got all usable multiples in it, and a set of to do multiple nodes, nodes that have to be still checked, are built to each starting node. In each cycle iteration those sets are updated for each starting node separately, each starting node gets one turn to check one multiple node. At the end of each iteration, the algorithm checks whether there is a non-empty intersection of possible intersecting nodes and if such intersection exists, it checks whether there exist paths from this node to both starting nodes.

The above outlined algorithm for finding path intersection also very intensively depends on the existence of such intersection. So far we can not stop the algorithm without searching the entire index that is reachable from the two starting points. It obviously also suffers from the impossibility of telling the mutual position of two nodes in the indexing structure. Therefore the time computational complexity is unacceptably high when looking for a connection that apparently does not exist in a very large graph.

Summarization The outlined algorithms in this section are implemented with an emphasis to represent the idea of searching paths using the designed index to RDF graphs. Thus they provide a vast space for further optimization, for example to speed the first algorithm up a cycle could be used instead of the approach with recursion. Also the access structures to the indexing structure could be improved to save significant amount of time to speed the algorithms up. Another way to improve the indexing structure is to create a second level that would ease the problem of mutual position of nodes in the graph. This approach is further discussed in the next section.

5 Future work

As it was proposed in the previous section, the future work will focus on two directions of improving this project. Firstly, we will work on optimization of the implemented algorithms to get better results on large scale data.

Secondly, as it was recognized in the previous section the main drawback of both algorithms is that it is not able to tell the mutual position of two nodes in the graph. So the future work in this direction is to solve this problem. Under heavy research is a second level indexing structure on the designed index. The idea is that the signatures, representing transformed stand-alone components of the original RDF graph will be assumed as nodes and the edges will be the connections through the multiple nodes to other signatures. Then the same transformation as it is presented in this paper will be applied again.

6 Concluding remarks

In comparison to the index structure designed in [5], the path between two nodes, if it exists, does not have to be precomputed to create the index. Instead, the index is used to compute the path or the ρ *connection* between two nodes. One of pros of the former indexing structure is that once all paths among all nodes are computed, the searching of the ρ path is very fast and effective. The cons are that such index is very memory intensive and the time to create such index is great. The index structure discussed in this paper can be created in linear time and the storage complexity is also linear to the size of the original RDF graph.

As was mentioned in Section 1 we recognize three ρ operators but the demonstrated indexing structure together with its algorithms can handle only two of them. The future work thus will also focus on extending the index to provide search for all Semantic Associations defined by the ρ operators.

The aim of this project is to create a scalable indexing structure for RDF graphs accompanied with algorithms providing the ρ operators functionality with acceptable time and space computational complexity. In present time the designed indexing structure provides solid base for such work but new approaches mentioned in previous sections has to be taken into account to make it all possible.

References

1. Kemafor Anyanwu and Amit Sheth. The rho operator: Discovering and ranking on the semantic web. 2003.
2. D. Brickley and R. V. Guha. Resource description framework schema specification. 2000.
3. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *The 11th Intl. World Wide Web Conference (WWW2002)*, 2002.
4. O. Lassila and R. R. Swick. Resource description framework: Model and syntax specification. 1999.
5. Agarwal Minal, Gomadam Karthik, Krishnan Rupa, and Yeluri Durga. Rho: Semantic operator for extracting meaningful relationships from semantic content.
6. T.Grust. Accelerating xpath location steps. In *The 11th Intl. World Wide Web Conference (WWW2002)*, pages 109–120, 2002.
7. Sanjeev Thacker, Amit Sheth, and Shuchi Patel. Complex relationships for the semantic web. 2001.
8. Pavel Zezula, Giuseppe Amato, Franca Debole, and Fausto Rabitti. Tree signatures for XML querying and navigation. *Lecture Notes in Computer Science*, 2824:149–163, 2003.