A Path-Oriented RDF Index for Keyword Search Query Processing

Paolo Cappellari[1], Roberto De Virgilio[2], Antonio Maccioni[2], and Mark Roantree[1]

[1]School of Computing
Dublin City University, Dublin, Ireland
{pcappellari,Mark.Roantree}@computing.dcu.ie
[2]Dipartimento di Informatica e Automazione
Universitá Roma Tre, Rome, Italy
{dvr,maccioni}@dia.uniroma3.it

Abstract

Most of the recent approaches to keyword search employ graph structured representation of data. Answers to queries are generally sub-structures of the graph, containing one or more keywords. While finding the nodes matching keywords is relatively easy, determining the connections between such nodes is a complex problem requiring on-the-fly time consuming graph exploration. Current techniques suffer from poorly performing worst case scenario or from indexing schemes that provide little support to the discovery of connections between nodes.

In this paper, we present an indexing scheme for RDF that exposes the structural characteristics of the graph, its paths and the information on the reachability of nodes. This knowledge is exploited to expedite the retrieval of the sub-structures representing the query results. In addition, the index is organized to facilitate maintenance operations as the dataset evolves. Experimental results demonstrates the feasibility of our index that significantly improves the query execution performance.

1 Introduction

In 2006, the Linked Open Data initiative (http://linkeddata.org/) inspired practitioners, organizations and universities to either publish or build from scratch RDF (Resource Description Framework, a graph-oriented logical data model) datasets from data that had previously been stored using traditional models [1], contributing to the definition of what is today called the *Web of Data*. The main objectives in searching this web of data are: the location and retrieval of results that are most relevant to the user's search, and to give more relevance to valid results currently missed (or low ranked) by modern search engines. The methodology for achieving these aims is to include as much semantics as possible in datasets and in general, RDF has been used to provide indexes with rich semantics to better interpret user queries.

In a scenario where the online data is constantly increasing, the difficulty for users is locating and retrieving the data that accurately meet their requirements. Having to know how data is organized and the query language to access data represent an obstacle to information access to non expert users. For this reason, keywords search systems are increasingly popular. Many approaches (e.g. [4, 8, 10, 12, 14, 15]) implement information

retrieval (IR) strategies on top of traditional database systems, with the goal of eliminating the need for users to understand query languages or be aware the data organization. A general approach involves the construction of a graph-based representation where query processing addresses the problems of an exhaustive search over the RDF graph. Two main steps are involved in the approach. Firstly, the system retrieves those parts of the graph that match users keywords with a subsequent identification of any connections across graph segments returned by the query process. Secondly, the system ranks the *combined* graph segments and top-k results are presented from the candidate result set. Clearly, the graph search is a crucial step. Typically it is supported by indexing systems (computed off-line) to guarantee the efficiency of the search. Existing systems [9, 13, 18] focus mainly on indexing nodes information (e.g. labels, position into the graph, cardinality and so on) to achieve scalability and to optimize space consumption. While locating nodes matching the keywords is relatively efficient using these indexes, determining the connections between graph segments is a complex and time-consuming task that must be solved on-the-fly at query processing time. To provide results in a reasonable time, current approaches do not perform an exhaustive search. As index schemes do not adequately support the discovery of connections between nodes, heuristics must be introduced to assist the search in locating a more complete result set. Thus, while the introduction of RDF-based solutions offers simple user interfaces with genuine semantic search features, the problem now lies with how to associate or connect intermediate result sets and how to manage the cost of determining those associations.

Contribution. We present a novel indexing scheme for RDF datasets that captures associations across RDF paths *before* query processing and thus, provides both an exhaustive semantic search and superior performance times. Unlike other approaches involving implementation based solutions, we follow a process that starts with the definition of the index at a conceptual level. It comprises paths and information on the reachability of nodes within the RDF graph. The next step is a logical translation for a relational database. Eventually, at the physical level, we choose optimization techniques based on physical indexing and partitioning. Because of the storage model foundations, the system can easily represent structural aspects of an RDF graph. Moreover we provide a set of procedures to insert or delete nodes (resources) and edges (properties) into the index and thus, support updates to the RDF graph. Such index is deployed in YAANII [5], a novel keyword search framework that leverages a joint use of scoring functions and solution building algorithms to get the best results for the initial result set. Experiments demonstrate how the proposed indexing scheme allows to significantly improve the efficiency of the overall process. The paper is organized as follows: Section 2 discusses related work; Section 3 introduces the basic concepts and illustrates how we model our path-oriented index; based on this model, Section 4 describes how the index is built and maintained in an efficient manner; finally Section 5 provides experimental evaluations and in Section 6, conclusions and future work are presented.

2 Related Work

Several proposals [9, 13, 18] implement in-memory structures that focus on node indexing. Others [16, 19] focus on indices for SPARQL query execution and join efficiency, not offering concrete support to graph exploration for keyword search. In [9], authors

provide a *Bi-Level INdexing Keyword Search* (BLINKS) method for the data graph. This approach assumes keywords can only occur in nodes, not in edges, and is based on pre-computed distances between nodes. The system implements two indices: one index stores information on which nodes are reachable from a given node; the other index is a hash table storing the shortest distance between pairs of nodes. In [13], authors propose a linked-list indexing scheme for RDF. The index is composed of a dictionary table, a statement table and a resource table. The dictionary table maintains the association between each resource and its (generated) identifier. Primarily, this table acts as reverse look-up identifier to resource label, to complete the answer to a query. The statement table maintains the list of the $\langle s, p, o \rangle$ RDF statements, where each statement has three references, each pointing to the next statement using the same s, p, o respectively. The resource table contains information about all the resources (s and o), linking each one to the first statement in which it occurs, and collecting statistics about statements presenting such resources. In [18], authors propose an approach to keyword search in RDF graph through query computation, implementing a system called SEARCHWEBDB. It is supported by two index structures: a keyword-to-element index and a graph index. The former implements an inverted index to associate each possible keyword to nodes or edges in the graph. To capture semantically similar words such as synonyms, every term is expanded with its similar term as described in WordNet [6]. The latter stores schema information of the graph, that is classes and relations between classes. The authors refer to this type of schema as a *summary graph*. Contrary to those approaches that index the entire graph, SEARCHWEBDB derives the query structure by enriching the summary graph with the input keywords. The search and retrieval process for the *enriched* summary graph, with all its possible distinct paths beginning from some keyword elements, provides a set of queries that once calculated, provides the final sub-graph answers. Other proposals focus on indexing graph substructures (e.g. paths, frequent subgraphs, trees). Typically, these indexes are exploited in approaches dealing with graph matching problems, often to filter out graphs that do not match an input query. Approaches in this area can be classified in: graph indexing and subgraph indexing. In graph indexing approaches, e.g. gIndex [20], TreePi [21], FG-Index [3], the graph database consists of a set of small graphs. The indexing aims at finding all database graphs that contain or are contained by a given query graph. On the other hand, subgraph indexing approaches, e.g. GraphGrep [7], TALE [17], GADDI [22], aims at indexing large database graph, with the goal of finding all (or a subset of) the subgraphs that match a given query efficiently. Our indexing scheme provides an agile solution with respect to graph substructures indexing approaches and enriches traditional node/edges indexing proposals with exhaustive information about connections between nodes.

3 Index Modeling

Our goal is to model a generic indexing scheme to support queries execution and keyword based search engines. Generally, standard queries are composed of a set of, possibly, interrelated path expressions. The result to a standard query is the portion of the graph that matches the path-expression(s) provided in the query specification. In the keyword based search paradigm, users are assumed to be agnostic of the schema and the query is a list of keyword terms. In keyword search systems the focus is on discov-

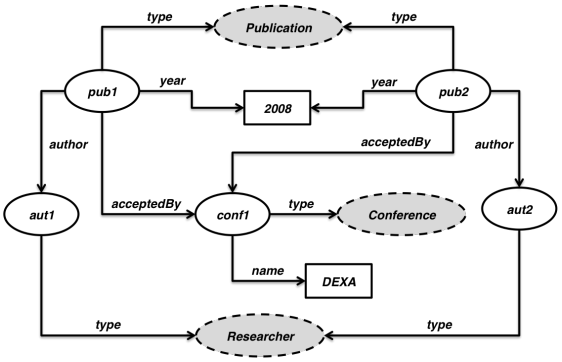


Fig. 1. An example of reference

ering connections between nodes matching keywords, on top of which query answers are built (as subgraphs). We would support systems that perform standard queries, such as path expressions, as well as keyword search, where the emphasis is on discovering connections between those parts of the graph holding information relevant to the input keywords. In particular, we focus on semantic dataset expressed in RDF format, that is a model that describes a directed labeled graph, where nodes are resources (identified by URIs) and edges have a label that expresses the type of connection between nodes.

Definition 1 *A **labeled directed graph** G is a three element tuple G = {V, L, E} where V is a set of nodes, L is a set of labels and E is a set of edges of the form* $e(v, u)$ *where* $v, u \in V$ *and* $e \in L$.

In G we call *sources* the nodes v_{src} with no incoming edges (i.e. $\nexists e(u, v_{src}) \in E$), and *sinks* the nodes v_{sink} with no outgoing edges (i.e. $\nexists e(v_{sink}, u) \in E$). We call *intermediate node*, a node that is neither a source nor a sink. Consider the example in Fig. 1. It illustrates an ontology about *Publications* written by *Researchers* (i.e. authors) accepted and published into a *Conference*. We have two sources, `pub1` and `pub2`, and four sinks, of which for instance we have `Publication` and `Conference`.

From the data graph point of view, in a RDF graph we have classes and data values (i.e. sinks), URIs (i.e. intermediate nodes and sources) and edges. Since it can be assumed the user will enter keywords corresponding to attribute values such as a name rather than using a verbose URI (e.g. see [18]), keywords might refer principally to edges and sinks of the graph. Therefore in our framework we are interested to index each path starting from a source and ending into a sink. Moreover, any node can be reached by at least one path originating from one of the sources. Paths originating from sources and reaching sinks includes (sub-)paths that stop in intermediary nodes. In case a source node is not present, a fictitious one can be added. To this aim, the so-called *Full-Path* is a path originating in a source and ending in a sink.

Definition 2 (Full-Path) *Given a graph* $G = \{V, L, E\}$*, a full-path is a sequence* $pt = v_1\text{–}e_1\text{–}v_2\text{–}e_2\text{–}\ldots - e_{n-1}\text{–}v_f$ *where* $v_i \in V$*,* $e_i \in L$ *(i.e.* $e_i(v_i, v_{i+1}) \in E$*),* v_1 *is a source and the final node* v_f *is a sink. We refer to* v_i *and* e_i *as tokens in* pt.

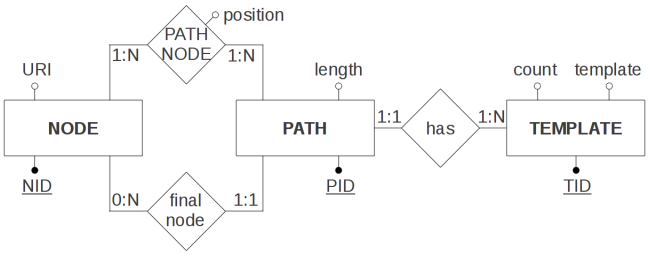


Fig. 2. Conceptual modeling of the index

In Fig. 1 a full-path pt_k is `pub1-author-aut1-type-Researcher`. The *length* of a path corresponds to the number of its nodes; the position of a node corresponds to its position in the presentation order of all nodes. In the example, pt_k has length 3 and the node `aut1` is in position 2. In the rest of the paper we refer to paths as full-paths.

The sequence of edge labels (i.e. e_i) describes the structure of a path. In some sense, the sequence of e_i is a *schema* for the information instantiated in the nodes. We can say that paths sharing the same structure carry homogeneous information. More properly, we say that the sequence of e_i in a path represents its *template*. Given a path pt its template t_{pt} is the path itself where each node v_i in pt is replaced with the wild card #. In our example pt_k has the following template: `#-author-#-type-#`. Several paths can share the same structure: it allows us to cluster paths according to the template they share. For instance the path pt_j `pub2-author-aut2-type-Researcher` has the same template of pt_k, that is pt_j and pt_k are in the same cluster.

In our framework we follow a three levels modeling. Starting from conceptual level, Fig. 2 shows an ER-diagram modeling the major constructs in our index. We start from the entity NODES modeling a node in the graph. The label is characterized by the attribute *URI* and each node is identified by *NID*. We then have the main entity PATHS representing a full-path. In particular, each path is identified by *PID*, presents the *length*, and the relation *final node* with NODES, representing the sink of the path. Each node *belongs* to a path with a *position* (i.e. the relation *pathnode*). Finally we have the entity TEMPLATES identified by *TID*, presenting the sequence of edge labels (i.e. the attribute *template*) and the number of paths sharing the template (i.e. the attribute *count*). Between PATHS and TEMPLATES a one-to-many relation assigns a template to each path.

At logical level, we have a straightforward transformation to a relational model. Fig. 3 shows the logical modeling of Fig. 2, populated with data from the example in Fig. 1. Each entity is transformed into a relation with the corresponding primary key and with foreign keys for the one-to-many relationship it contains. The many-to-many relationship PATHNODES is also transformed to a relation, correlating paths with their nodes, and vice versa, through the attribute *position*.

As RDF datasets are often very large, at physical level we exploit relational DBMS features to tune our schema for better performance. In particular we employ Oracle as relational DBMS. First of all we implement horizontal partitioning on the tables, based on the value of a column (called range). In particular PATHS is partitioned with respect to the template (i.e. *TID*). In this way each partition is a cluster of homogeneous full-paths.

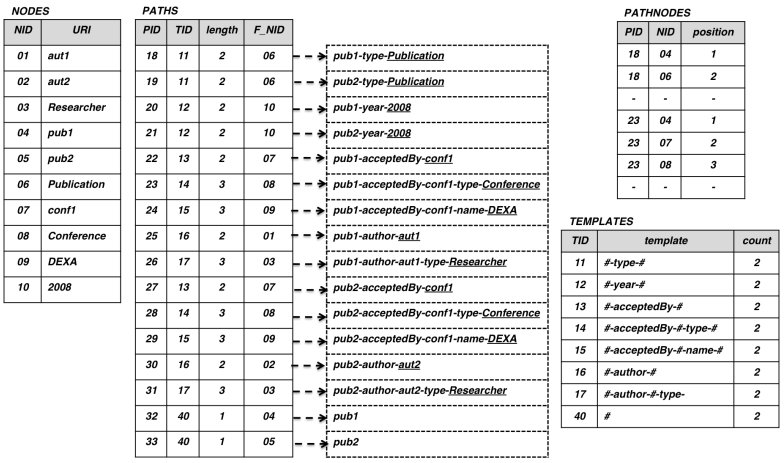


Fig. 3. Logical Modeling of the index

Then we define physical indexes on the single partitions and on the other unpartitioned tables. Specifically, we employ the Oracle Index-organized tables (IOTs), that are a special style of table structure stored in a B-tree index frame. Along with primary key values, in the IOTs we also store the non-key column values. IOTs provide faster access to table rows by the primary key or any key that is a valid prefix of the primary key. Because the non-key columns of a row are present in the B-tree leaf block itself, there is no additional block access for index blocks. In our case PATHS and PATHNODES are organized as IOTs. The matching is supported by standard IR engines (c.f. Lucene Domain index (LDi)[1]) embedded into Oracle as a type of index. In particular we define a LDi index on the attributes *label* and *template* of tables NODES and TEMPLATES respectively. Further, semantically similar entries such as synonyms, hyponyms and hypernyms are extracted from WordNet [6], supported by LDi.

With this broad goal in mind, our indexing system provides two major advantages: (i) intersections between paths, needed to build the subgraph solutions, are efficiently identifiable (as we will show in the next section), and (ii) the template based classification splits the information from the graph into non-overlapping subsets, as each cluster represents an instance of a different template (or schema).

4 Index Management

In this section, we describe the index creation, discuss the processes to maintain the the index when the graph changes and, finally, illustrate how to query the index.

4.1 Constructing the Graph Index

Given a graph, the creation of the index requires three steps: (i) node hashing, (ii) source identification, and (iii) computation of full-paths, from source to sink.

[1] http://www.scribd.com/doc/38406372/Lucene-Domain-Index

Algorithm 1: BFS-based graph exploration algorithm

Input : Sets $triples$ and $Roots$
Output: PATHS, PATHNODES and TEMPLATES populated

```
1  foreach r ∈ Roots do
2      Queue ← ∅ ;
3      pt_r ← NewID() ;
4      PATHS ← PATHS ∪ ⟨pt_r , NewID() , 1, r ⟩ ;
5      PATHNODES ← PATHNODES ∪ ⟨pt_r , r , 1⟩;
6      Enqueue(Queue , pt_r );
7      while Queue is not empty do
8          Dequeue (Queue,pt) ;
9          if ∃⟨pt , τ , l , n ⟩ ∈ PATHS then
10             foreach ⟨n , p , o ⟩ ∈ triples do
11                 if ∄⟨pt' , _, _, o ⟩ ∈ PATHS then
12                     pt' ← NewID() ;
13                     τ' ← NewTemplate(τ, p ) ;
14                     PATHS ← PATHS ∪ ⟨pt' , τ , l + 1, o ⟩ ;
15                     foreach ⟨m , pt , pos ⟩ ∈ PATHNODES do
16                         PATHNODES ← PATHNODES ∪ ⟨m , pt' , pos ⟩ ;
17                     PATHNODES ← PATHNODES ∪ ⟨o , pt' , l + 1⟩ ;
18                     Enqueue (Queue, pt' ) ;
```

The input dataset is assumed to be an RDF graph where the information is modeled as a set of triples. Each triple has the form $\langle s, p, o\rangle$, where: s is the *origin resource* (known as *subject*), o is the *target resource* (*object*), p is a property linking the origin with the target resource (known as *property*). In the first step, we populate columns URI and NID ub table NODES with nodes URIs and IDs, respectively. In the second step, we identify the source nodes, that is all subjects of triples that never occur as objects. Finally in a third step, we explore the graph to build the full-paths: once identified, the path is decompose to populate tables TEMPLATES, PATHS and PATHNODES. We explore the graph using a concurrent version of the *Breath-First Search* (BFS) algorithm, shown in Algorithm 1. Starting from each source (line 1), the algorithm creates a one node length path pt_r composed of the source node only, associates the path with an ID using the function $newID$, and puts pt_r in the $Queue$ (lines 3-6). While there are elements in the queue (line 7), the algorithm dequeue pt from $Queue$ (line 8): pt represents the path just indexed and ending into the node n. Starting from n the algorithm collects all nodes o, object into triples $\langle n, p, o\rangle$ (line 10). If a path ending in o does not exist (line 11), then new paths pt' extending pt with the edge $\langle n, p, o\rangle$ (lines 13-17) are created. Finally, it enqueues pt', that will possibly create new paths of length $l + 2$.

4.2 Index Maintenance

When the dataset changes, the index must be updated as a consequence. We developed a number of basic maintenance operations: insertion/deletion of a node; insertion/deletion

of an edge; and update of the content of a node/edge label. More sophisticated operations, such as the insertion or removal of a triple or a set of triples, can be implemented as a composition of these basic functions.

Before presenting the algorithms for these basic operations, it is useful to introduce a new concept and a new routine that simplify the discussion of the maintenance operations: the *tailpath* and the *forward-concatenation* . A tailpath is a sub-path of an full-path. Specifically, it is the sub-path starting at an intermediate position and ending with the final node of the full-path (i.e. a sink).

Definition 3 (TailPath) *A tailpath is represented as a pair* $\langle nodes, sub\text{-}template \rangle$ *where* $nodes$ *is a list of pairs* $\langle n, pos \rangle$*,* n *is a node belonging to the tailpath and* pos *its position. As an extension,* $sub\text{-}template$ *is the corresponding template of tailpath.*

Tailpaths are not stored in the index, but computed dynamically if and when needed for maintenance operations.

Definition 4 (Forward-Concatenation) *The forward-concatenation function creates new paths by merging a selected path with a tailpath.*

The forward-concatenation function uses a new edge, connecting the last node of the selected path, with the first node of the tailpath to form a new Full-Path.

Algorithm 2 illustrates how to compute a tailpath starting from a node at a specific position in a given path pt. The tailpath is built by querying the information from tables

Algorithm 2: TailPath.

Input : A Path pt, an integer pos
Output: A Tail Path $tailPath$

```
1 set ← ∅ ;
2 foreach ⟨pt , n_r , pos ′⟩ ∈ PATHNODES : pos ′ ≥ pos do
3     set ← set ∪ ⟨n_r , pos ′ − pos + 1⟩ ;
4 τ_sub ← SubTemplate(pt, pos ) ;
5 tailPath ← ⟨set , τ_sub ⟩;
6 return tailPath;
```

PATHNODES (lines 2-3) and TEMPLATES (line 4) from our index. To compute the sub-template we rely on a simple function, $sub\text{-}template$ that retrieves the template for a path and returns its substring starting at the specified position (where the position is determined by the # symbols in the template). As an example, refer to Fig. 1 and its index in Fig. 3, with the tailpath for path with ID 23 (i.e. `pub-acceptedBy-conf1-type-Conference`) starting at position 2 (i.e. node `conf1`) is `conf1-type-Conference`.

Algorithm 3 describes the forward-concatenation mechanism to append a set of tailpaths $tailPaths$ to an existing path pt, where an edge p connecting the last node of the path with the first node of the tailpath is provided. In the algorithm, for each tailpath (line 1) a new path pt_{new} is created (lines 2-4). The function $JoinTemplates$

Algorithm 3: ForwardConcatenation.

Input : A set $tailPaths$ of tailpaths, a path pt to extend, an edge p
Output: The tables PATHS, PATHNODES, and TEMPLATES updated.

```
1 foreach ⟨pairs , τ_sub ⟩ ∈ tailPaths do
2     pt_new ← NewID();
3     τ_new ← JoinTemplates(τ, τ_sub );
4     ⟨n_max , pos_max ⟩ ← MaxPos(pairs ) ;
5     PATHS ← PATHS ∪ ⟨pt_new , τ_new , n_f + pos_max , n_max ⟩;
6     foreach ⟨pt , n , pos ⟩ ∈ PATHNODES do
7         PATHNODES ← PATHNODES ∪ ⟨pt_new , n , pos ⟩;
8     foreach ⟨m , pos ⟩ ∈ pairs do
9         PATHNODES ← PATHNODES ∪ ⟨pt_new , m , pos + l ⟩;
```

concatenates two templates, updates the table TEMPLATES with the newly created template, and returns the identifier of such template. When updating TEMPLATES, if the template is already defined then no row is added to the table; the count value for the existing template is incremented and its identifier returned. Otherwise, a new row for the new template is inserted in the table. The function $MaxPos$ is used to retrieve the pair $\langle n, pos\rangle$ with highest position value from the set of nodes belonging to the tailpath. With this information defined, we can store the new path in PATHS (line 5).

Let us provide an example. Assume pt =`pub1-acceptedBy-conf1`, $tailPaths$ as {`DEXA`} and p as `name`. The new path is pt_{new} = `pub1-acceptedBy-conf1-name-DEXA`, with template `#-acceptedBy-#-name-#`. To complete the definition, we must associate the newly created path with its nodes. Nodes belonging to pt_{new} are: those belonging to pt (lines 6-7), and those belonging to the tailpath (lines 8-9).

Edge deletion. The edge to be deleted is specified as a parameter to the Algorithm 4 in the form of an RDF triple $\langle s, p, o\rangle$. This operation requires deleting all paths that

Algorithm 4: Delete an edge.

Input : A $triple\langle s , p , o \rangle$ representing the edge with label p between nodes s and o.
Output: The updated index.

```
1 foreach ⟨pt , n_o, pos ⟩ ∈ PATHNODES do
2     foreach ⟨pt , n_s, pos − 1⟩ ∈ PATHNODES do
3         if p = PropByPos(pt, pos-1) then
4             TailPaths ← TailPaths ∪ TailPath(pt, pos );
5             PATHS ← PATHS ∖ ⟨pt , _, _, _⟩;
6             PATHNODES ← PATHNODES ∖ ⟨pt , _, _⟩;
7             DelTemplate(pt);
8 if ∄⟨pt , n_o, _⟩ ∈ pathnodes then
9     ForwardConcatenation(TailPaths, ∅, ∅);
```

contain edge p. In our index, edges are not stored explicitly: they are encoded in the templates associated with the paths. In order to identify the paths containing p, we select those pt in which n_s, n_o (i.e. IDs associated to nodes s and o respectively) are directly connected (lines 1-2). Then, for each pt we verify if the edge between n_s and n_o is p. To this aim, we use the function $PropByPos$ that takes as input pt and the position $(pos - 1)$ of n_s (i.e. n_o is in position pos) and returns the edge outgoing from s (i.e. by accessing the template corresponding to pt).

Let us now give an example using the graph depicted in Fig. 1. Assume we want to remove property `acceptedBy` between `pub1` and `conf1` (i.e. corresponding to the triple $\langle pub1, acceptedBy, conf1\rangle$). The IDs of the paths having nodes `pub1` immediately preceding `conf1` are 22, 23 and 24 (see Fig. 3). Consider the path with ID 24; `conf1` occurring at position 2. Thus, $PropByPos$ (i.e. $PropByPos(24, 1)$) will access the template with ID 15 (i.e. `#-acceptedBy-#-name-#`) and return the sub-string `acceptedBy` between nodes in positions 1 and 2. Similarly we process also paths with IDs 22 and 23. Continuing the discussion of the algorithm, since n_o could become a source, we need to build the tailpaths from n_o (line 4) before deleting all pt. Thus we delete all pt and all corresponding nodes (lines 5-6). We must also update the corresponding count in TEMPLATES by using the function $DelTemplate$. If n_o became a source (line 8), we then invoke the forward-concatenation on the tailpaths (line 9) to build all paths from n_o.

Edge insertion. The input to Algorithm 5 is a triple $\langle s, p, o\rangle$ where p is the label for the new edge to add between (existing) resources s and o. We have to create the new

Algorithm 5: Insert a new edge

Input : A $triple\langle s\ , p\ , o\ \rangle$ representing the edge with label p between nodes s and o.
Output: The updated index.

```
1  TailPaths ← ∅ ;
2  foreach ⟨pt1 , n o, pos ⟩ ∈ PATHNODES do
3      TailPaths ← TailPaths ∪ TailPath(pt1, pos ) ;
4  foreach ⟨pt1 , τ , l , n s⟩ ∈ PATHS do
5      if ∄⟨pt1 , n o, _⟩ ∈ PATHNODES then
6          ForwardConcatenation(TailPaths, pt1, p) ;
7  foreach ⟨pt1 , no, 1⟩ ∈ PATHNODES do
8      PATHS ← PATHS ∖ ⟨pt1 , _, _, _⟩;
9      PATHNODES ← PATHNODES ∖ ⟨pt1 , _, _⟩;
10     DelTemplate(pt1) ;
```

paths involving p. To this aim we have to concatenate paths ending in s (i.e. node n_s) and tailpaths built from o (i.e. node n_o) as shown in (lines 2-6).

For instance, suppose we want to re-introduce the property `acceptedBy` we have removed in the previous example. The input triple is $\langle pub1, acceptedBy, conf1\rangle$. Only the path pt with ID 31 ends in `pub1`. The tailpaths from node `conf1` are: (1) sub-

path `conf1` with sub-template `#`, (2) sub-path `conf1-type-Conference` with sub-template `#-type-#`, and (3) `conf1-name-DEXA` with sub-template `#-name-#`. Connecting pt with such tailpaths through property `acceptedBy`, we obtain back paths with IDs 22, 23, 24. If n_o becomes a source, we delete all paths rooted in n_o (lines 7-10).

Remaining maintenance operations. The rest of the maintenance operations are simpler and intuitive, and are now discussed briefly. In *Node insertion*, we basically have to insert a new entry into NODES. Since it is not yet linked to other nodes, the new node is a source with an associated path (of length 1). Therefore, we must insert a path updating PATHS, PATHNODES and possibly, TEMPLATES. *Node deletion* is the inverse operation to node insertion. Assuming the node is not linked to any other, we have to delete one entry from tables: NODES, PATHS, PATHNODES and possibly, from TEMPLATES if the count in TEMPLATES reaches zero for the associated template. *Edge update* requires as input, the triple to identify which edge to update and a new value for such edge. Since edge information is encoded in templates, we must access and parse the template strings. Therefore, we retrieve the paths containing the input triple and we generate a new template for them with where the old value is replaced by the new one. In *Node update*, once identified the node, we only have to update its URI in table NODES.

Computational Complexity. In this section we present a discussion about the computational complexity of the creation and maintenance of our indexing scheme. Before commencing the discussion, let us introduce the notation we will use in the remainder of this section. Let R be the number of sources, E be the number of edges and V the number of vertexes. We indicate with PT the number of paths in the index. With PT_n we denote the number of paths containing a specific node n, and with $PT_{n_1,n_2,\ldots,n_k}$ the number of paths containing the node sequence $n_1, n_2, \ldots, n_k$. Finally, TP indicates the number of tailpaths in the forward concatenation and L the length of a path.

The Index Creation is $O(R \times (E + V))$. Such algorithm is an implementation of BFS (i.e. notoriously it has complexity $O(E + V)$) and it is invoked once for each source. Let us remark that our approach, initially, does not compute all the possible paths between a source and a node (leaf or intermediate), but only that ones ending into a sink: thus the complexity for the BFS is much lower than $O(E + V)$. Index update operations on a single node (e.g. insertion, deletion or update of a node), are trivial and it is straightforward to verify that they have complexity $O(1)$. The function Forward-Concatenation defined in Algorithm 3 is $O(TP \times L)$. In fact, it depends on how many tail paths (TP) we have to concatenate in the creation of the new path; and the creation of a new path depends on its length (L). In practical cases (also attested by experiments) L is rather smaller than TP. Therefore we can reformulate in $O(TP)$. The Algorithm 4 to delete an existing edge p connecting nodes s, n is $O(PT_{s,o} \times L)$. Deleting the path having the edge of interest implies to delete the occurring nodes (i.e. L). Since the paths containing the triple $\langle s, p, o\rangle$ are $PT_{s,o}$, deleting this information for all such paths spends $O(PT_{s,o} \times L)$. In case the node o becomes a source, then we also have the ForwardConcatenation on the tailpaths from o. Although this operation is merely a copy, its complexity is still $O(PT_{s,o} \times L)$. Summarizing, the overall complexity is $2\times(PT_{s,o}\times L) \in O(PT_{s,o}\times L)$. Since L is rather small, we have $O(PT_{s,o})$. The update of an edge in a path has complexity $O(PT_{s,o})$. We assume the function $ReplaceEdge$

```
Q1:
SELECT DISTINCT pn.PID
FROM PATHS AS pn
WHERE pn.F_NID in (
  SELECT NID
  FROM NODES AS n
  WHERE lcontains(URI,
        <inputTerm>) )
```

```
Q2:
SELECT pn.PID,
       pn.position
FROM PATHNODES AS pn
WHERE pn.NID =
      <inputNodeID>
```

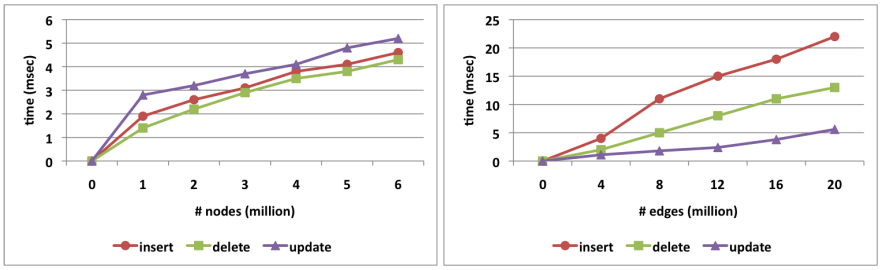
```
Q3:
SELECT pn1.PID
FROM PATHNODES AS pn1
WHERE pn1.NID in (
  SELECT pn2.NID
  FROM PATHNODES AS pn2
  WHERE pn2.PID =
        <inputPathID> )
```

Fig. 4. Sample queries on the index

to have complexity $O(1)$. Because the edge could belongs to many paths, $ReplaceEdge$ must be performed more times. Since the number of paths with such edge is $PT_{s,o}$, thus the complexity is $O(PT_{s,o})$. Algorithm 5 defines the insertion of a new edge p between two existing nodes s, o. The algorithm first takes the PT_o tailpaths from o: we have $O(PT_o \times L)$. Then it performs a forward-concatenation of the above tailpaths with all the paths ending in s (i.e. in the worst case PT_s paths). Thus the forward concatenation (i.e. $O(PT_o \times L)$) must be performed for each path ending in s; we conclude the overall complexity is $O(PT_s \times PT_o \times L)$. As we noticed before, in practical cases, L is rather small, which leads us to say that the complexity of the operation is $O(PT_s \times PT_o)$. In an extreme case we could have $PT_o + PT_s = PT$; in this situation most of the paths ends in o and/or s. Therefore $O(PT_o)$ or $O(PT_s) \in O(PT)$. In practical cases it is rare that the two nodes s, o concentrate all the paths, then $O(PT_o)$ or $O(PT_s) \ll O(PT)$.

4.3 Index Querying

In this section, we provide a few examples, shown in Fig. 4, on how to query our index. Query **Q1** retrieves all paths having at least one node matching the input keyword `inputTerm`. This query uses the function *lcontains*, part of the Oracle SQL syntax, that exploits the Lucene full-text search capability. Although we use Oracle, as described in Section 5, this query can easily be adjusted to suit other SQL dialects. In fact, many modern DBMS supports Lucene by implementing their own variation on the SQL syntax. The query limits its attention to the final node (F_NID) of the paths because keywords only occurs in final nodes, as intermediary nodes only contain URI references. In a variation, **Q1** can be join with TEMPLATES to perform keyword search on paths' template, i.e. searching for keywords on the edges. The second query, **Q2**, retrieves all paths containing a generic node `inputNodeID` along with its position in each path. The node can be any between a source, intermediary or sink. This information can be exploited when the position of a node is a critical information to perform further analysis, like: reachability of such node from other nodes, the computation of the paths starting from or ending in such node. Last query, **Q3**, retrieves all the paths intersecting with a given path `inputPathID`. It first locates the nodes belonging to `inputPathID`, then the paths sharing such nodes. In keyword search systems, this query is useful when searching for connections (on the graph) between the nodes (or edges) matching some keyword. A keyword search system builds a solution (i.e. a subgraph) assembling intersecting paths with nodes (or edges) matching some keyword.



(a) (b)

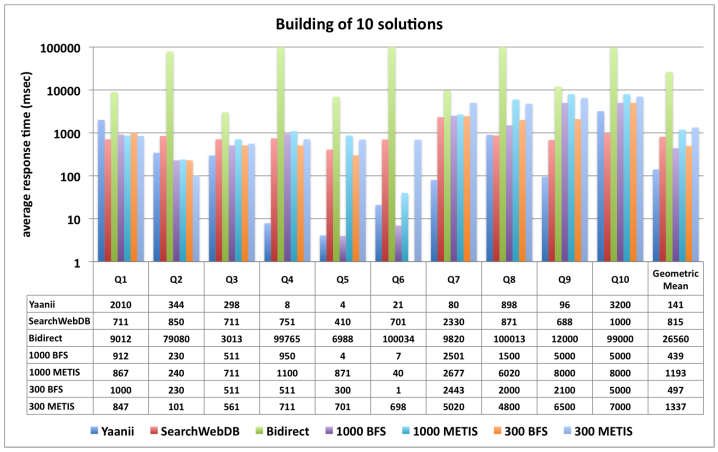
Fig. 5. Maintenance Scalability with respect to #nodes (a) and #edges (b)

5 Implementation

We implemented our path-oriented index into YAANII [5], a Java system for keyword search query over RDF datasets. All procedures for building and maintenance of our index have been serialized in PL/SQL operations and deployed in Oracle 11g v2. With the experiments discussed in this section we have measured how much the index improves query performance in YAANII. We used a widely-accepted benchmark of ten queries on DBLP for keyword search evaluation. DBLP (Digital Bibliography & Library Project, a computer science bibliography) is a dataset containing 27M triples about computer science publications. Due to space limitations we omit the query list here (see [11]). Experiments were conducted on a dual quad core 2.66GHz Intel Xeon, running Linux RedHat, with 8 GB of memory, 6 MB cache, and a 2-disk 1Tbyte striped RAID array. We evaluated the performance of: index building, index update and query execution. On top of DBLP we indexed roughly 17M of entries into the table PATHS, 28M into PATHNODES, 0,6M into TEMPLATES and 6M into NODES. The building task took a total 37 hours, and includes: the import of dataset from a single file encoding DBLP in NTRIPLE, and the execution of the BSF algorithm to compute the fullpaths. The final disk space required by the storage of the index was 718MB.

Fig. 5 collects the performance of maintenance operations (i.e. insertion, deletion and update of nodes and edges). The figure reports the scalability of such operations with respect to the increasing number of nodes (i.e. Fig. 5.(a)) and edges (i.e. Fig. 5.(b)). At each group of nodes (e.g. $1, 2, \ldots, 6$ millions) or edges (e.g. $4, 8, \ldots, 27$ millions), we inserted and updated 100 nodes (edges) and then we deleted them. Then we measured the average response time (ms) for one node (or edge). Opposite to the building step, the maintenance of the index follows good performance, satisfying practical scenarios with frequent updates of the dataset.

For query execution evaluation, we integrated our index in YAANII and we compared performance with the most related approaches: SEARCHWEBDB [18], bidirectional search [11] (we refer to it as BIDIRECT) and the several techniques based on graph indexing, i.e. 1000 BFS, 1000 METIS, 300 BFS, 300 METIS (see details in [11]). We ran the queries in [11] ten times and measured the average response time. Precisely, the total time of each query is the time for computing the top-10 answers. The query runtimes are shown in Figure 6. In general BIDIRECT performs poorly, SEARCHWEBDB is comparable with BFS and better than METIS. YAANII with our index implemented is



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Geometric Mean
Yaanii	2010	344	298	8	4	21	80	898	96	3200	141
SearchWebDB	711	850	711	751	410	701	2330	871	688	1000	815
Bidirect	9012	79080	3013	99765	6988	100034	9820	100013	12000	99000	26560
1000 BFS	912	230	511	950	4	7	2501	1500	5000	5000	439
1000 METIS	867	240	711	1100	871	40	2677	6020	8000	8000	1193
300 BFS	1000	230	511	511	300	1	2443	2000	2100	5000	497
300 METIS	847	101	561	711	701	698	5020	4800	6500	7000	1337

Fig. 6. Response Times

the best (on average) among the cited approaches: it was consistently the best for most of the queries; it outperforms all competitors by a large margin, improving times by nearly a factor varying from 3 to 188 in the geometric mean. As demonstrated in [2], the complexity of YAANII algorithm outperforms other approaches, but by employing our index the entire process, requiring frequent queries similar to **Q1**, **Q2** and **Q3** discussed in Section 4.3, speed-up significantly.

6 Conclusion and Future Work

The web of data is a powerful mechanism for both users and organisations but due to its size (and the size of many individual information components) provides a significant challenge when searching for information needs. In this paper, we presented a path-oriented indexing scheme for the large graph structures that contain the semantic datasets comprising the web of data. The key feature of the index is that it exposes the structural characteristics of the graph: its paths, the structure (schema) of these paths, and the information on the reachability of nodes. By exploiting this information, we can expedite query execution, especially for keyword based query systems, where query results are built on the basis of connections between nodes matching keywords. As graph exploration is a complex and time consuming task, usually computed on-the-fly during query processing, our index facilitates a far more efficient query process. We developed a prototype system by integrating our index into YAANII, a system for keyword searching query RDF using path computations. Results show that the index significantly increases the performance of YAANII, outperforming other approaches while still providing the desired, exhaustive search. Current research is focused on: an investigation into mathematical properties to weight relevant paths and templates with respect to the graph; a more compact index and compression technique to reduce space consumption; and further optimizations for both index creation and maintenance.

References

1. C. Bizer and R. Cyganiak. D2R server: Publishing relational databases on the semantic web. In *Proc. of ISWC*, 2006.
2. Paolo Cappellari, Roberto De Virgilio, Antonio Maccioni, and Michele Miscione. Keyword based search over semantic data in polynomial time. In *Proc. of ICDE Workshops*, pages 203–208, 2010.
3. James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *Proc. of SIGMOD*, pages 857–872, 2007.
4. Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. Xsearch: a semantic search engine for xml. In *Proc. of VLDB*, pages 45–56, 2003.
5. Roberto De Virgilio, Paolo Cappellari, and Michele Miscione. Cluster-based exploration for effective keyword search over semantic datasets. In *Proc. of ER*, pages 205–218, 2009.
6. C. Fellbaum, editor. *WordNet: an electronic lexical database*. MIT Press, 1998.
7. Rosalba Giugno and Dennis Shasha. Graphgrep: A fast and universal method for querying graphs. In *Proc. of ICPR*, pages 112–115, 2002.
8. Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: ranked keyword search over xml documents. In *Proc . of SIGMOD*, pages 16–27, 2003.
9. Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: ranked keyword searches on graphs. In *Proc. of SIGMOD*, pages 305–316, 2007.
10. Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proc. of VLDB*, pages 850–861, 2003.
11. Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proc. of VLDB*, pages 505–516, 2005.
12. Benny Kimelfeld and Yehoshua Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proc. of PODS*, pages 173–182, 2006.
13. Dave Kolas, Ian Emmons, and Mike Dean. Efficient linked-list rdf indexing in parliament. In *5th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2009.
14. Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *Proc. of SIGMOD*, pages 563–574, 2006.
15. Alexander Markowetz, Yin Yang, and Dimitris Papadias. Reachability indexes for relational keyword search. In *Proc. of ICDE*, pages 1163–1166, 2009.
16. Thomas Neumann and Gerhard Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 3(1):256–263, 2010.
17. Yuanyuan Tian and Jignesh M. Patel. Tale: A tool for approximate large graph matching. In *Proc. of ICDE*, pages 963–972, 2008.
18. Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *Proc. of ICDE*, pages 405–416, 2009.
19. Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
20. Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *Proc. of SIGMOD*, pages 335–346, 2004.
21. Shijie Zhang, Meng Hu, and Jiong Yang. Treepi: A novel graph indexing method. In *Proc. of ICDE*, pages 966–975, 2007.
22. Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proc. of EDBT*, pages 192–203, 2009.