RDF Keyword Search Query Processing via Tensor Calculus

Roberto De Virgilio

Dipartimento di Informatica e Automazione
Universitá Roma Tre, Rome, Italy
dvr@dia.uniroma3.it

Abstract. Keyword-based search over (semi)structured data is considered today an essential feature of modern information management systems and has become an hot area in database research and development. Answers to queries are generally sub-structures of a 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 propose an indexing scheme for RDF graphs based on the first principles of linear algebra, in particular on *tensorial calculus*. Leveraging our abstract algebraic framework, our technique allows to expedite the retrieval of the sub-structures representing the query results.

1 Introduction

Today, many organizations and practitioners are all contributing to the "Web of Data", building RDF repositories either from scratch or by publishing in RDF data stored in traditional formats. In this scenario, keywords search systems are increasingly popular. Many approaches 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. Typically it is supported by indexing systems (computed off-line) to guarantee the efficiency of the search. Existing systems [4,6] 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.

Related Work. Several proposals [4,6] implement in-memory structures that focus on node indexing. In [4], 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. In [6], authors propose an approach to keyword search in RDF graph through query computation,

R. Meersman et al. (Eds.): OTM 2012, Part II, LNCS 7566, pp. 780–788, 2012.

implementing a system called SEARCHWEBDB. It 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: graph and subgraph indexing. In graph indexing approaches, e.g. TreePi [7], 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. Subgraph indexing approaches, e.g. TALE [5], aims at indexing large database graphs, to find all (or a subset of) the subgraphs that match a given query efficiently.

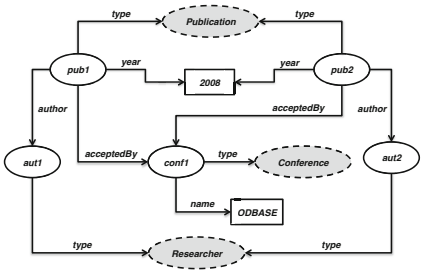
Contribution. In this paper we propose a novel approach based on first principles derived from the linear algebra field. We present a novel indexing scheme for RDF datasets that captures associations across RDF paths *before* query processing and thus, provides both an exhaustive semantic search and superior performance times. Unlike other approaches involving implementation based solutions, we exploit the first principles of linear algebra, in particular on *tensorial calculus*. Leveraging our abstract algebraic framework, our technique allows to expedite the retrieval of the sub-structures representing the query results. Because of the algebra 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. The paper is organized as follows: Section 2 illustrates how we model our path-oriented index; based on this model, Section 3 describes how the index is built and maintained in an efficient manner; finally Section 4 provides experimental evaluations and Section 5 sketches some conclusions and future work.

2 Index Modeling

This section is devoted to the definition of a general model capable of representing all aspects of a given RDF graph. Our overall objective is to give a rigorous definition of a RDF graph, along with few significant properties, and show how such representation is mapped within a standard tensorial framework.

Path Oriented Modeling. Our goal is to model a generic indexing scheme to support queries execution and keyword based search engines. 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. To this aim, we briefly recall (and simplify) some notions in [3].

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* $e \in L$.



(a)

Nodes	
1	aut1
2	aut2
3	Researcher
4	pub1
5	pub2
6	Publication
7	conf1
8	Conference
9	ODBASE
10	2008

Full-paths	
1	pub1-type-Publication
2	pub2-type-Publication
3	pub1-year-2008
4	pub2-year-2008
5	pub1-acceptedBy-conf1-type-Conference
6	pub1-acceptedBy-conf1-name-ODBASE
7	pub1-author-aut1-type-Researcher
8	pub2-acceptedBy-conf1-type-Conference
9	pub2-acceptedBy-conf1-name-ODBASE
10	pub2-author-aut2-type-Researcher

Templates	
1	#-type-#
2	#-year-#
3	#-acceptedBy-#-type-#
4	#-acceptedBy-#-name-#
5	#-author-#-type-#

(b)

Fig. 1. An example of reference: (a) RDF graph and (b) nodes, full-paths and templates

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 [6]), 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.*

In Fig. 1.(a) 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 occurrence 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. Fig. 1.(b) presents the entire sets of nodes, full-paths and templates. We used a table-like representation where we assigned an id-number to each entry. Finally we provide the *intersection* between paths. Given two paths pt_1 and pt_2, there is an intersection between pt_1 and pt_2, denoted by $pt_1 \leftrightarrow pt_2$, if $\exists v_i \in pt_1$ and $\exists v_j \in pt_2$ such that $v_i = v_j$.

Tensorial Representation. Let us define the set $\mathcal{P}$ as the set of all full-paths in a RDF graph G, with $\mathcal{P}$ being finite. A *property* of a path p is defined as an application $\pi : \mathcal{P} \rightarrow \mathit{\Pi}$, where $\mathit{\Pi}$ represents a suitable property codomain. Therefore, we define the application of a property $\pi(p) := \langle \pi, p \rangle$, *i.e.*, a property related to a path $p \in \mathcal{P}$ is defined by means of the pairing path-property; a property is a surjective mapping between a path and its corresponding property value. A *RDF graph index* ($\mathcal{I}$) is defined as the product set of all paths, and all the associated properties. Formally, let us introduce the family of properties π_i, $i = 1, \ldots, k + d < \infty$, and their corresponding sets $\mathit{\Pi}_i$; we may therefore model a RDF graph index as the product set

$$\mathcal{I} = \underbrace{\mathcal{P} \times \mathit{\Pi}_1 \times \ldots \times \mathit{\Pi}_{k-1}}_{\mathbb{B}} \times \underbrace{\mathit{\Pi}_k \times \ldots \times \mathit{\Pi}_{k+d}}_{\mathbb{H}} . \tag{1}$$

The reader should notice as we divided explicitly the first k spaces $\mathcal{P}, \mathit{\Pi}_1, \ldots, \mathit{\Pi}_{k-1}$, from the remaining ones. We have two different categories: *body* ($\mathbb{B}$) and *head* ($\mathbb{H}$). It should be hence straightforward to recognize the tensorial representation of $\mathcal{I}$:

Definition 3 (Tensorial Representation). *The* tensorial representation *of a RDF graph index $\mathcal{I}$, as introduced in equation* (1)*, is a multilinear form $\mathcal{I} : \mathbb{B} \longrightarrow \mathbb{H}$.*

A RDF graph index can be therefore rigorously denoted as a rank-k tensor with values in $\mathbb{H}$. Now we can model some codomains $\mathit{\Pi}$ with respect to the conceptual modeling provided above. In particular we define the property $\mathit{\Pi}_1$ as the set $\mathcal{N}$ of all nodes into the RDF graph, $\mathit{\Pi}_2$ as the set of positions $\mathcal{O}$ of the nodes in $\mathcal{N}$ belonging to the paths in $\mathcal{P}$ (*i.e.* $\mathcal{O} \subseteq \mathbb{N}$), $\mathit{\Pi}_3$ as the set of lengths $\mathcal{L}$ of full-paths in $\mathcal{P}$ (*i.e.* $\mathcal{L} \subseteq \mathbb{N}$) and $\mathit{\Pi}_4$ as the set of all templates $\mathcal{T}$ associated to the paths in $\mathcal{P}$. In the next paragraph we will illustrate how we organize such properties in $\mathbb{B}$ and $\mathbb{H}$. As a matter of fact, properties may *countable* or *uncountable*. By definition, a set A is *countable* if there exists a function $f : A \rightarrow \mathbb{N}$, with f being injective. For instance the sets $\mathcal{P}$, $\mathcal{N}$, $\mathcal{T}$ are countable while $\mathcal{L}$ and $\mathcal{O}$ are uncountable. Therefore countable spaces can be represented with natural numbers and we can introduce a family of injective functions called *indexes*, defined as: $\mathrm{idx}_i : \mathit{\Pi}_i \longrightarrow \mathbb{N}, i \in [1, k + d]$. When considering the set $\mathcal{P}$, we additionally define a supplemental index, the *path index function* $\mathrm{idx}_0 : \mathcal{P} \rightarrow \mathbb{N}$, consequently completing the map of all countable sets of $\mathcal{I}$ to natural numbers.

Implementation. Our treatment is general, representing a RDF graph with a tensor, *i.e.*, with a multidimensional matrix, due to the well known mapping between graphs and matrices (cf. [1]). However, a matrix-based representation need not to be *complete*. Hence, our matrix effectively requires to store only the information regarding connected nodes in the graph: as a consequence, we are considering sparse matrices [2], *i.e.*, matrices storing only non-zero elements. For simplicity's sake, we adopt the *tuple notation*. This particular notation declares the components of a tensor $M_{i_1 i_2 \ldots i_k}(\mathbb{F})$ in the form

$$\begin{pmatrix}
\cdot & \cdot & \cdot & (1,2,1) & \cdot & (2,2,1) & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & (1,2,1) & (2,2,1) & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & (1,2,2) & \cdot & \cdot & \cdot & \cdot & \cdot & (2,2,2) \\
\cdot & \cdot & \cdot & \cdot & (1,2,2) & \cdot & \cdot & \cdot & \cdot & (2,2,2) \\
\cdot & \cdot & \cdot & (1,3,3) & \cdot & \cdot & (2,3,3) & (3,3,3) & \cdot & \cdot \\
\cdot & \cdot & \cdot & (1,3,4) & \cdot & \cdot & (2,3,4) & \cdot & (3,3,4) & \cdot \\
(2,3,5) & \cdot & (3,3,5) & (1,3,5) & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & (1,3,3) & \cdot & (2,3,3) & (3,3,3) & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & (1,3,4) & \cdot & (2,3,4) & \cdot & (3,3,4) & \cdot \\
\cdot & (2,3,5) & (3,3,5) & \cdot & (1,3,5) & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}$$

Fig. 2. Representative matrix of a RDF graph

$\mathcal{M} = \left\{ \{i_1 i_2 \dots i_k\} \to f \neq 0 \,, f \in \mathbb{F} \right\}$, where we implicitly intended $f \neq 0 \equiv 0_{\mathbb{F}}$. As a clarifying example, consider a Kroneker vector $\delta_4 \in \mathbb{R}^5$: its sparse representation will be therefore constituted by a single tuple of one component with value 1, *i.e.*, $\{ \{4\} \to 1 \}$. In our case we have the notation $\mathbb{B} \longrightarrow \mathbb{H} = \mathbb{B}(\mathbb{H})$ where we consider $\mathbb{B} = \mathcal{P} \times \mathcal{N}$ and $\mathbb{H} = \mathcal{O} \times \mathcal{L} \times \mathcal{T}$.

Let us consider the example pictured in Fig. 1: the corresponding representative matrix is depicted in Fig. 2. For typographical simplicity, we omitted $0_{\mathbb{H}} = (0,0,0)$, denoted with a dot. For clarity's sake, we outline the fact that $\mathcal{I}$ is a rank-2 tensor with dimensions 10 (i.e. the rows corresponding to the paths), 10 (i.e. the columns corresponding to the nodes). In fact, we have ten full-paths and ten nodes in $\mathcal{P}$ and $\mathcal{N}$ sets, respectively. Then we map the elements of $\mathcal{P}$, $\mathcal{N}$ and $\mathcal{T}$ by using the index functions idx_0, idx_1 and idx_4 (*i.e.* let us consider the index numbers shown in Fig. 1). In the sparse representation we have $\{\{1,4\} \to (1,2,1), \{1,6\} \to (2,2,1), \dots, \{10,5\} \to (1,3,5)\}$. For instance, referring to Fig. 1.(b), the row 1 corresponds to the path `pub1-type-Publication` with nodes `pub1` (*i.e.* column 4) and `Publication` (*i.e.* column 6). The values $(1,2,1)$ and $(2,2,1)$ indicate that `pub1` and `Publication` are in positions 1 and 2, respectively, and the length of the path is 2 and the corresponding template `#-type-#` has index 1.

3 Index Management

This section is devoted to the exemplification of our conceptual method, with the applicative objective of analyzing RDF data. With reference to the example provided in Section 2 (*i.e.* Fig. 1), in the following we simplify our notation, employing a matrix M_{ij}: i referring to the index of paths, and j being the index of nodes.

Processing Queries. Usually keyword search query requires real-time processing of the information organized in the index to access efficiently to the needed sub-structure for composing the final solutions. In this case we have to perform different processing of the index. A ***Node Query*** (NQ) finds all paths containing a given node n. In other words, given a keyword we would find all paths containing nodes matching such keyword. Therefore, given $j = \mathrm{idx}_1(n)$, we build a Kroneker vector as a vector δ_j, with $|\delta_j| = |\mathcal{N}|$, and finally apply of the rank-2 tensor represented by M_{ij} to δ_j, *i.e.* $r = M_{ij}\delta_j$. For instance, referring to our example, let us consider the node *Researcher*,

corresponding to the third column of the matrix (*i.e.* $j = \mathrm{idx}_1(Researcher) = 3$). In this case $\delta_3 = \{\{3\} \to 1\}$. Consequently, the resulting vector will be $r = M_{ij}\delta_3 = \{\{7\} \to (3,3,5), \{10\} \to (3,3,5)\}$, that are the paths with indexes 7 and 10. For our purpose, we are interested in particular to retrieve paths ending into a given node n. We call such query *Final Node Query* (FNQ). In this case we have to impose a condition to filter the result of NQ, that is all paths containing the node n such that the position of n corresponds to the length of the path. To this aim we apply the condition to r employing the map operator, as follows $\widetilde{r} = \mathrm{map}(\mathrm{cond}, r)\,, \mathrm{cond} : \mathbb{B} \times \mathbb{H} \longrightarrow \mathbb{F}$. The cond function, as indicated above, is a map between properties of $\mathcal{I}$ and a suitable space $\mathbb{F}$, *e.g.*, natural numbers for a boolean result. In our case $\mathrm{cond} = \{o(\cdot) == l(\cdot)\}$. The reader should notice a shorthand notation for an implicit boolean function: as for all descendant of the C programming language, such definition yields 1 if the condition is met, *i.e.*, the position o of the node n is equal to the length l of the path, 0 otherwise. Furthermore, we recall the fact that being M_{ij} represented as a sparse matrix, the application of our condition is well defined, being applicable to the stored values. For instance if we consider again the node *Researcher*, since we have $(3,3,5)$ the condition is satisfied and we have the same result (*i.e.* the rows 7 and 10). Similar to NQ, we can define the ***Path Query*** (PQ) that retrieves all nodes belonging to a given path p. Given $i = \mathrm{idx}_0(p)$, we build a Kroneker vector as a vector δ_i, with $|\delta_i| = |\mathcal{P}|$, and apply of the rank-2 tensor represented by M_{ij} to δ_i, *i.e.*, $r = M_{ij}\delta_i$. From our example, if the path is that with index 1, we have $\delta_1 = \{\{1\} \to 1\}$ and $r = M_{ij}\delta_1 = \{\{4\} \to (1,2,1), \{6\} \to (2,2,1)\}$, that are the nodes with indexes 4 and 6. A more interesting query is the ***Path Intersection Query*** (PIQ). Given two paths p_1 and p_2, PIQ verifies if there exists an intersection between p_1 and p_2. Therefore given $i = \mathrm{idx}_0(p_1)$ and $k = \mathrm{idx}_0(p_2)$, we build two Kroneker vectors as δ_i and δ_k, with $|\delta_i| = |\delta_k| = |\mathcal{P}|$, and for each vector we apply of the rank-2 tensor, *i.e.* $r_1 = M_{ij}\delta_i$ and $r_2 = M_{kj}\delta_k$. Finally the query is easily performed by exploiting the tensor application coupled with the Hadamard product: $r = r_1 \circ r_2$ and p_1 and p_2 present an intersection *iff* $r \neq \emptyset$. For instance, referring to our example, the paths with indexes 1 and 2 (*i.e.* the first two rows of our matrix) the Hadamard product results $r = \{\{6\} \to (2,2,1)\}$ that is the node with index 6 in common (*i.e. Publication*). Similar to PIQ, a ***Path Intersection Retrieval Query*** (PIRQ) retrieves all paths having an intersection with a given path p. In this case we have to execute a PQ on p resulting r as the set of all nodes in p. Then for each node n in r we execute a NQ on n. The union of results from each NQ represents all paths having an intersection with p. For instance, the path with index 7 presents three nodes (*i.e.* 1, 3 and 4). Therefore, executing a NQ for each node, we have that only the columns 3 and 4 have non-zero values in other rows (*i.e.* rows 1, 3, 5, 6 and 10) that are the paths with an intersection. Retrieving paths, sometimes we need only parts of that paths. To this aim cutting operations are needed. Therefore we introduce the ***Path Cutting Query*** (PCQ) that given a path p and a node n returns the part of p *starting* from n or *ending* into n. In this case given $i = \mathrm{idx}_0(p)$ and $j = \mathrm{idx}_1(n)$, we have $pos = o(M_{ij})$ and $r = \mathrm{map}(o(\cdot)\ \theta\ pos, M_{ij})$. The operator θ can be $\geqslant$ or $\leqslant$ according as we need to start from n or ending into n, respectively. Of course we can consider also parts of a path limited by two nodes: we have to combine the two conditions. For instance consider the row 7:

if we would extract the part of the path starting from the node *aut1*, we have to impose the condition $o(\cdot) \geqslant 2$ resulting the nodes with indexes 1 and 3.

Maintenance Queries. An important purpose of the index is the maintenance, *i.e.*, when the dataset changes, the index must be updated as a consequence. In this paragraph we provide a number of basic maintenance operations expressed in our conceptual model: 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. As a matter of fact, in our computational framework insertion and deletion procedures are reflected by *assigning values* in a given tensor. Removing a value is comfortably implemented by assigning the null value to an element, *i.e.*, $M_{ij} = (0, 0, 0)$, and analogously, inserting or modifying a value is attained via $M_{ij} = (o, l, t)$. The ***Node Deletion*** (ND) procedure has to delete all paths involving the given node n. Therefore we execute a NQ to retrieve r (*i.e.* all paths involving n) and for each index i corresponding to a path p (*i.e.* $i = \text{idx}_0(p)$) in r we have $\forall j : M_{ij} = (0, 0, 0)$. The *Node Insertion* (NI) procedure extends the matrix with a new column, *i.e.* $M_{i\ j+1}$, given the node n generates $k = \text{idx}_1(n)$ and $\forall i : M_{ik} = (0, 0, 0)$. The ***Edge Deletion*** (ED) procedure deletes all paths containing a given edge e. In this case we can have two types of input: only the label of e or a triple $\langle n_1, e, n_2 \rangle$ (*i.e.* we assume n_1 and n_2 existing into the graph). In the former case, we select all templates t' containing the label e and we delete all paths corresponding to t'. Therefore $k = \text{idx}_4(t')$ and $r = \text{map}(t(\cdot) = k, M_{ij})$. Then for each path index i extracted from r we have $\forall j : M_{ij} = (0, 0, 0)$. In the latter case we have to identify all paths p containing e, n_1 and n_2 where n_1 and n_2 are directly connected (*i.e.* the distance between the positions of n_1 and n_2 is 1). Therefore given r generated as previously described, we execute two NQs on r: we obtain r_1 and r_2 executing NQ with n_1 and n_2 as inputs, respectively. Then we compute the Hadamard product: $r_3 = r_1 \circ r_2$. Finally we use the map to extract all paths such that the distance between the positions of n_1 and n_2 is 1: given $j_1 = \text{idx}_1(n_1)$ and $j_2 = \text{idx}_1(n_2)$ then $\widetilde{r}$ = $\text{map}(o_{j_2}(\cdot) - o_{j_1}(\cdot) == 1, r_3)$. All values in the rows corresponding to the paths in $\widetilde{r}$ will be set to $(0, 0, 0)$. For instance, assume we would remove the property *acceptedBy* between *pub1* and *conf1*. In this case we have to delete two paths: rows 5 and 6. We retrieves that paths because the associated templates (*i.e.* 3 and 4) contain the property and the nodes are directly connected (*i.e.* difference between positions is 1). Finally we set all values on that rows to $(0, 0, 0)$. The ***Edge Insertion*** (EI) procedure has to insert a new edge e. It is the more complex maintenance operation. The input is in term of a triple $\langle n_1, e, n_2 \rangle$ (*i.e.* we assume n_1 and n_2 existing into the graph). In this case we have to generate new paths involving e from that of the matrix. To this aim we have to retrieve all paths p_1 containing n_1 and of which we keep the part ending in n_1 (*i.e.* $\widetilde{r_1}$ with NQ and PCQ) and all paths p_2 containing n_2 and of which we keep the part starting from n_2 (*i.e.* $\widetilde{r_2}$ with NQ and PCQ). Finally we have to *concatenate* each row in $\widetilde{r_1}$ with all rows in $\widetilde{r_2}$ to generate the new paths to insert into M, as follows:

- for each path (*i.e.* row i) of $\widetilde{r_1}$
  - we calculate the new template. We modify the old template t, inserting e, and we generate t' (*i.e.* a trivial string operation). Then we have $k = \text{idx}_4(t')$;
  - we calculate the (max) position z_1 of the last node, that is: $z = \max_{o_i}(\widetilde{r_1})$;

- we calculate z_2 and z_3 of the first and last node in each row j in $\widetilde{r_2}$;
- we calculate the new lengths l' of the paths to generate from the row i in $\widetilde{r_1}$ with the rows j in $\widetilde{r_2}$, that is $l' = z_1 + (z_3 - z_2)$;
- we update length and template of each tern in the row i of $\widetilde{r_1}$ with l' and k;
- we update position, length and template of each triple-value in the rows j of $\widetilde{r_2}$ with o', l' and k, where $o' = o_i(\widetilde{r_2}) - (z_2 - z_1)$;

– for each new path we generate a new row in the matrix.

Let us suppose we would re-introduce the property *acceptedBy* between *pub1* and *conf1*. In this case we have $\widetilde{r_1} = \{\{1\} \rightarrow (1,2,1), \{3\} \rightarrow (1,2,2), \ldots\}$ that are all paths with non-zero values on the column 4. In this case all the extracted parts correspond to one node, *i.e.*, `pub1`. Analogously, $\widetilde{r_2}$ contains all paths with non-zero values on the column 7 considering only the terns where the position is greater than 2. They correspond to `conf1-type-Conference` and `conf1-name-ODBASE`. Now we have to concatenate *pub1* with all paths in $\widetilde{r_2}$ obtaining the paths `pub1-acceptedBy-conf1-type-Conference` and `pub1-acceptedBy-conf1-name-ODBASE`. The ***Node/Edge Update*** (NU/EU) procedures do not execute any operation on the matrix (*i.e.* they refer to the labels).

4 Experimental Results

We implemented our index into YAANII [3]. All procedures for processing and maintenance the index are linked to the Mathematica 8.0 computational environment. Experiments were conducted on a dual core 2.66GHz Intel Xeon, running Linux RedHat, with 4 GB of memory, 6 MB cache, and a 2-disk 1Tbyte striped RAID array.

We evaluated the performance of our index in terms of index building and index querying. We used a widely-accepted benchmark on DBLP for keyword search evaluation. It is a dataset containing 26M triples about computer science publications. Moreover we used iMDB (a portion of 6M triples) and LUBM (50 Universities corresponding to 8M triples). Table 1 shows the results. We report the number of full-paths computed (*i.e.* 17M for DBLP, 3M for iMDB and 5M for LUBM). Then t_{paths} and $t_{importing}$ measure the times to calculate the paths (*i.e.* minutes) and to import them into our matrix (*i.e.* seconds) in the Mathematica environment. Both the tasks are executed by Mathematica scripts. In particular we calculate the paths by using the BFS procedure. It produces a text file directly imported by mathematica to build the matrix. The table shows efficient performance. A more significant results is the memory usage (*i.e.* MB). iMDB and LUBM require limited memory (suitable for a mobile device) while DBLP requires more memory resources. However we can split the matrix into chunks suitable also for memory-constraint devices. Once we created the matrix, we store it as a text

Table 1. Index Performance

	#paths	t_{paths}	$t_{importing}$	**Mem**	**Size**	**NQ**	**FNQ**	**PQ**	**PIQ**	**PIRQ**	**PCQ**	**ND**	**NI**	**ED**	**EI**
DBLP	17M	10m	113s	544	182	12	13	9	11	36	1	13	0,001	33	41
iMDB	3M	2m	21s	108	32	2,1	2,5	1	1,5	11	1	3	0,001	9	13
LUBM	5M	4m	43s	181	53	3,5	4	2	3	15	1	4	0,001	13	18

file on the file system. The size of such file (*i.e.* MB) is feasible in any dataset. Finally we executed several processing and maintenance queries on the index (*i.e.* they are measured in terms of msec). For the processing times we ran the queries ten times and measured the average response time. In any dataset such queries require few msec. For the maintenance times we inserted and updated 100 nodes (edges) and then we deleted them. Then we measured the average response time for one node (or edge). Also the maintenance operations provide good performance.

5 Conclusion and Future Work

In this paper, we presented a path-oriented indexing scheme for the RDF graphs based on the first principles of linear algebra. By exploiting this scheme, 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. Current research is focused on: an investigation into mathematical properties to weight relevant paths and templates with respect to the graph, and further optimizations.

References

1. Bondy, A., Murty, U.S.R.: Graph Theory. Springer (2010)
2. Davis, T.A.: Direct Methods for Sparse Linear Systems. SIAM (2006)
3. De Virgilio, R., Cappellari, P., Miscione, M.: Cluster-Based Exploration for Effective Keyword Search over Semantic Datasets. In: Laender, A.H.F., Castano, S., Dayal, U., Casati, F., de Oliveira, J.P.M. (eds.) ER 2009. LNCS, vol. 5829, pp. 205–218. Springer, Heidelberg (2009)
4. He, H., Wang, H., Yang, J., Yu, P.S.: Blinks: ranked keyword searches on graphs. In: SIGMOD, pp. 305–316 (2007)
5. Tian, Y., Patel, J.M.: Tale: A tool for approximate large graph matching. In: ICDE (2008)
6. Tran, T., Wang, H., Rudolph, S., Cimiano, P.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In: ICDE, pp. 405–416 (2009)
7. Zhang, S., Hu, M., Yang, J.: Treepi: A novel graph indexing method. In: ICDE (2007)