

Incremental characterization of RDF Triple Stores

Adrien Basse, Fabien Gandon , Isabelle Mirbel, Moussa Lo

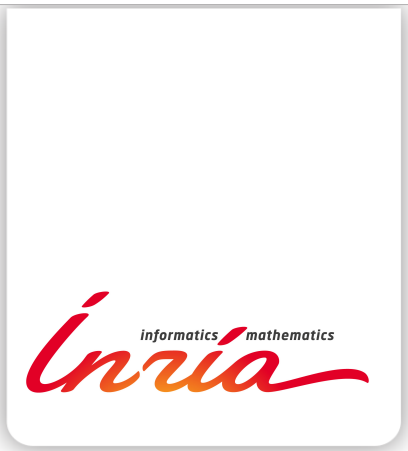
RESEARCH REPORT

N° 7941

April 2012

Project-Team Wimmics

ISSN 0249-6399 ISRN INRIA/RR--7941--FR+ENG



Incremental characterization of RDF Triple Stores

Adrien Basse *†, Fabien Gandon *, Isabelle Mirbel‡, Moussa Lo †

Project-Team Wimmics

Research Report n° 7941 — April 2012 — 21 pages

Abstract: Many semantic web applications integrate data from distributed triple stores and to be efficient, they need to know what kind of content each triple store holds in order to assess if it can contribute to its queries. We present an algorithm to build indexes summarizing the content of triple stores. We extended Depth-First Search coding to provide a canonical representation of RDF graphs and we introduce a new join operator between two graph codes to optimize the generation of an index. We provide an incremental update algorithm and conclude with tests on real datasets.

Key-words: RDF, graph mining, indexation, DFS coding

* INRIA Méditerranée, France
† LANI UFR SAT, University Gaston Berger, Senegal
‡ I3S Labotary (CNRS, UNS), France

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Incremental characterization of RDF Triple Stores

Adrien Basse, Edelweiss, INRIA Méditerranée, France
Fabien Gandon, Edelweiss, INRIA Méditerranée, France
Isabelle Mirbel, I3S Labotary (CNRS, UNS), France
Moussa Lo, LANI UFR SAT, University Gaston Berger, Senegal

ABSTRACT
Many semantic web applications integrate data from distributed triple stores and to be efficient, they need to know what kind of content each triple store holds in order to assess if it can contribute to its queries. We present an algorithm to build indexes summarizing the content of triple stores. We extended Depth-First Search coding to provide a canonical representation of RDF graphs and we introduce a new join operator between two graph codes to optimize the generation of an index. We provide an incremental update algorithm and conclude with tests on real datasets.

Keywords: RDF, graph mining, indexation, DFS coding

INTRODUCTION
Many semantic web applications face the problem of integrating data from distributed RDF (http://www.w3.org /RDF) triple stores. Several solutions exist for distributed query processing (Battle & Benson, 2008 ; Stuckenschmidt et al., 2004) and SPARQL 1.1 Federation (http://www.w3.org/2007/05/SPARQLfed/) defines extensions to the SPARQL (http://www.w3.org/TR/rdf-SPARQL-query) Query Language to support distributed query execution. These extensions allow us to formulate a query that delegates parts of the query to a series of services but one issue remains: how to automate the selection of triple stores containing relevant data to answer a query. This is especially true in the context of the Linking Open Data where numerous and very heterogeneous datasets are interlinked allowing interesting queries across several sources. To decompose and send queries targeting only relevant stores, we need a means to describe each store: an index structure which provides a complete and compact description of the content of the triple store.
Imagine you are interested in Tim Berners-Lee's activities and want to find his publications and some of his personal information. Figure 1 shows RDF graphs from two independent datasets: DBPedia (http://dbpedia.org) and DBLP (http://www4.wiwiss.fu-berlin.de/dblp/). Knowing what kind of knowledge is maintained by each store allows us to conclude that we have to combine publication information from DBLP with personal information from DBpedia. Figure 1 also illustrates what kinds of content one may frequently encounter in each dataset and therefore what

kind of knowledge one can expect to gain when accessing these datasets. Therefore, automatically identifying descriptive content for a triple store is a key problem.

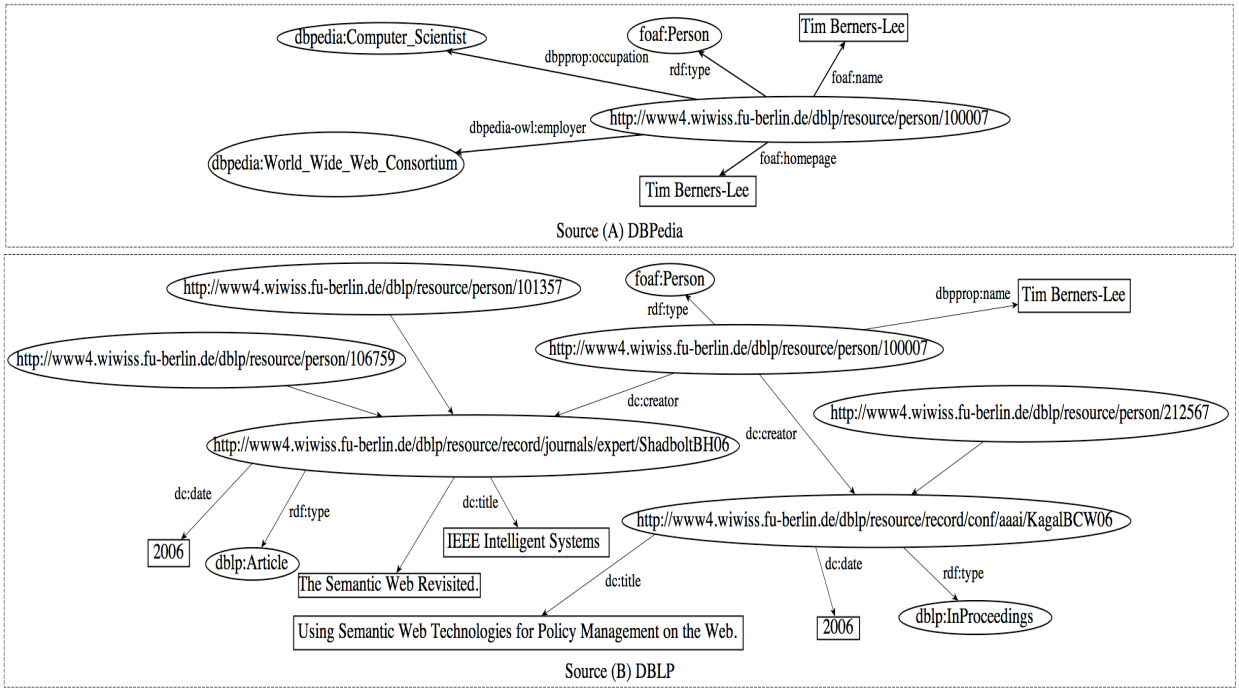


Figure 1. RDF graphs from of DBPedia and DBLP dataset.

To build indexes summarizing the content of triple stores and to use this indexes to guide distributed query processing we are interested in structure of SPARQL query. According to the information kept in the indexes, (Harth et al., 2010) and (Umbrich et al., 2011) classify the approaches adressing the problem of selection relevant sources. Some approaches (Inverted URI Indexing approaches) use as indexed items the URIs in the source. Others approaches (Schema-level Indexing approaches) use as indexed items the properties and/or URIs of classe of nodes in the source. Finally, there is a family of approaches (Multidimensional Histograms approaches and QTree approaches) which « combine description of instance and schema-level element ». The approach we propose belongs to family of Schema-level Indexing approaches and uses particular graphs as indexed items. Chosing graph as indexed items instead of uri, node and/or property or triple allow us to keep the structure of the information itself and in the futur to be able to decompose the distributed query into graph pattern (basic or group pattern graph (eg. http://www.w3.org/TR/sparql11-query/)) to determine relevant sources.
Section 2 describes our indexed item and its canonical representation nammed DFSR (Depth-First Search coding for RDF) code. Section 3 details the induction algorithm to build the different levels of the index structure. Section 4 discusses results of experiments. Section 5 presents an incremental algorithm to update the index structure when changes occur in the triple store. Section 6 surveys related works.

INDEXED ITEM AND DFSR CODING

Figure 2 shows an RDF graph describing people and containing cycles, blank nodes and multityped resources. We will use this example to explain our indexed item and its canonical representation. For the sake of readability we omit namespaces in the remaining paper (rdf :type instead of http://www.w3.org/1999/02/22-rdf-syntax-ns#type for instance).

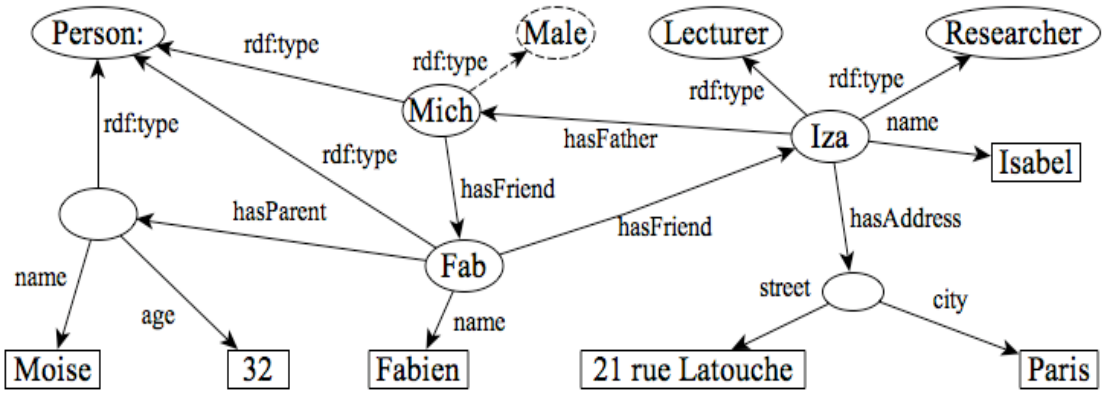


Figure 2. Example of RDF store.

Following the definition of graphs in (Baget et al., 2008) and (Harth et al., 2010) we propose the following definitions to describe and explain our indexed items (Infered RDF graph pattern).

Definition 1: *RDF triple, RDF graph*. Given U a set of URI with optional fragment identifier at the end, L a set of plain and typed Literal and B a set of blank nodes, a RDF triple is a 3-tuple $(s, p, o) \in (U \cup B) \otimes U \otimes B)$. *s* is the node subject of the RDF triple, *p* the predicate of the triple and *o* the node object of the triple. A RDF graph is a set of RDF triple.

Definition 2: RDF typed triple, RDF untyped triple. A RDF typed triple is a 3-tuple (s, p, o) ∈ (U∪B)⊗ (rdf :type) ⊗B). A RDF untyped triple is a 3-tuple (s, p, o) ∈ (U∪B) ⊗ (U\rdf: type) ⊗B).

Definition 3 : Type of node. In a RDF graph G, if a node *n* is subject of one RDF typed triple *(n, rdf:type, t)* then its type or class is the object (*t*) of this RDF typed triple. If a node *n* is subject of many RDF typed triples $\{(s, rdf :type, t_i), 1<i<n \}$ then we define its (conjunctive) type as the intersection of all object (t_i, $1<i<n$) of this RDF typed triples and note it as $t_1 \wedge t_2 \ldots \wedge t_n$.

Definition 4 : Infered RDF triples (IRDF triples). A Infered RDF triples of a RDF untyped triple *(s, p, o)* is the set of RDF triples $\{(s,p,o)\} \cup \{ (s, rdf :type, t_i), 1<i<n\} \cup \{ (o, rdf :type, c_j), 1<j<m\}$. To ensure that each node has at least one type we give by default the type *rdf :resource* to each node.

Definition 5 : Infered RDF graph (IRDF graph). A RDF graph *I* is the Infered RDF graph of a RDF graph *G* if and only if *I* is the union of all IRDF triples of the RDF untyped triples of G.

Definition 6 : Infered RDF graph pattern (IRDF graph pattern), instance of a IRDF graph pattern. A RDF graph *P* is the infered RDF graph pattern of an IRDF graph *I* if there is a mapping function M such that

- M maps URI, blank node and litteral to blank node.
- *{(s,p,o)} ∪ { (s, rdf :type, t_i), 1<i<n} ∪ { (o, rdf :type, c_j), 1<j<m}* is an IRDF triples in *I* if and only if *{(M(s),p,M(o))} ∪ { (M(s), rdf :type, t_i), 1<i<n} ∪ { (M(o), rdf :type, c_j), 1<j<m}* is an IRDF triples in *P*.

If *P* is the IRDF graph pattern of a RDF graph *G*, we also say that *G* is an instance of *P*.

Intuitively, to obtain an IRDF graph pattern from an IRDF graph *G* we replace in *G* the nodes of RDF untyped triples and subject of RDF typed triples with blank nodes.

Definition 7 : Size of IRDF graph pattern. The size of an IRDF graph pattern *I* is its number of untyped RDF triples or the number of its IRDF triples.

Throughout the paper we use a linear textual notation and a graphical notation form to represent IRDF graph pattern. A node of a untyped triple is labelled by its type and its label. The URI of nodes are replaced by character * to represent blank nodes. Figure 3 shows an example of IRDF graph pattern with instance in Figure 2 and its corresponding linear textual form and concise graphical notation form.

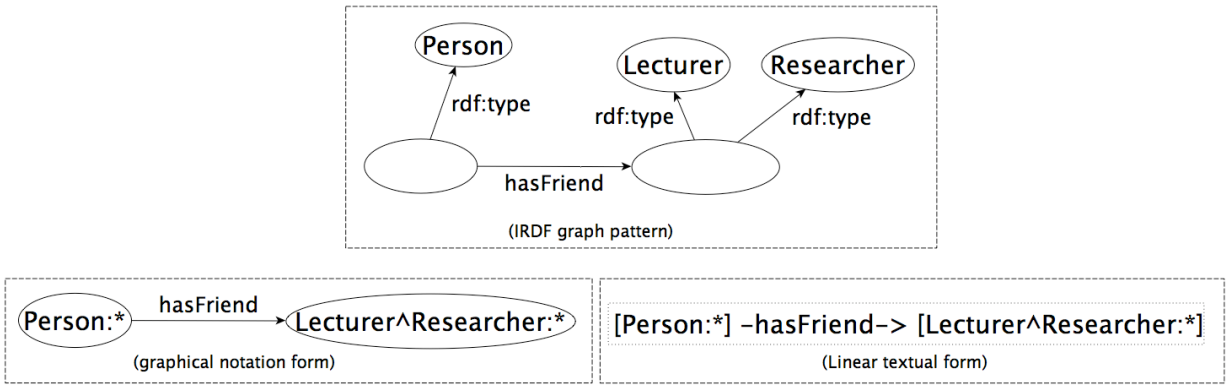


Figure 3. Graphical notation form and linear textual form.

To represent IRDF graph pattern in the index structure and improve the efficiency of some operations on graph as equality between graphs, isomorphism test, we use a canonical form of IRDF graph pattern. The canonical form proposed is an extention of Depth-First Search (DFS) coding of (Yan & Han, 2002) to the IRDF graph pattern. (Yan & Han, 2002) introduced a mapping of graphs to DFS codes. An edge *e* with $n(e)=(n_i, n_j)$ of an undirected labelled graph *G* is presented by a 5-tuple, $(i, j, l_G(n_i), t_G(e), l_G(n_j))$, where *i* and *j* denote the positions (DFS discovery times) of nodes n_i and n_j following a Depth-First Search. $l_G(n_i)$ and $l_G(n_j)$ are respectively the labels of n_i and n_j and $t_G(e)$ is the label of the edge between them. $i<j$ means n_i is discovered before n_j during the Depth-First Search. When performing a Depth-First Search in a graph, Yan and Han (2002) construct a DFS tree and defines an order. The forward edge set

contains all the edges in the DFS tree while the backward edge set contains the remaining edges. The forward edges are arranged in DFS order with their discovery times during the Depth-First Search. Two backward edges linked to a same node are arranged in lexicographic order. Given a node n_i, all of its backward edges should appear after the forward edge pointing to n_i. The sequence of 5-tuple based on this order is a DFS code. A graph may have many DFS codes and a DFS lexicographic order allows us to determine a canonical label called the minimum DFS code. (Han et al., 2007; Yan & Han, 2002, 2003; Yan et al., 2004) discuss DFS coding in the context of undirected labelled graphs. For directed labelled graphs (Maduko et al., 2008) captures the edge directions: in the 5-tuple $(i, j, l_G(n_i), t_G(e), l_G(n_j))$ if $i>j$ it means that $(l_G(n_i), t_G(e), l_G(n_j))$ is a backward edge. We adopted this coding to generate canonical labels for RDF graph patterns and called them DFSR codes.

Definition 8 : DFSR code. A DFSR code D of an IRDF graph pattern I is a sequence of 5-tuple such that for each untyped triple(s,p,o) in I corresponds a 5-tuple $(i, j, N(t_s), N(p), N(t_o))$ in D where t_s is the type of node s, t_o is the type of node o, N maps a string (type of node or property in I) to integer in such way that the resulting integer maintain lexicographical order of string, i (resp. j) denote the position of node s (resp. o) following a Depth-First Search in the set of RDF untyped triple of I.

Definition 9 : Size of DFSR code. The size of a DFSR code is the number of its 5-tuples.

An IRDF graph pattern may have many DFSR codes and we define a linear order in a set of DFSR to determine a canonical label of an IRDF graph pattern.

Definition 10 : *5-tuple order*. 5-tuple order is a linear order ($<_T$) in a 5-tuple set defined as follows. If $t_1 = (a_1,a_2,a_3,a_4,a_5)$ and $t_2 = (b_1,b_2,b_3,b_4,b_5)$ are two 5-tuples of integer then $t_1 <_T t_2$ if and only if $\exists\, i,\ 1 \le i \le 5$ such that $a_j = b_j$ where $j < i$ and $a_i \le b_i$.

Definition 11: DFSR order. DFSR order is a linear order ($<_D$) in a DFSR code set defined as follows. If $d_1 = (e_1,e_2,\ldots,e_n)$ and $d_2 = (f_1,f_2,\ldots,f_n)$ are two DFSR codes of size n with e_k and d_k $1 \le k \le n$ are 5-tuples, then $d_1 <_D d_2$ if and only if $\exists\, i,\ 1 \le i \le n$ such that $e_j = f_j, j<i$ and $e_i \le f_i$.

Definition 12: Minimum DFSR code. Given an IRDF graph I and its set of DFSR codes D, the minimum DFSR code of I is the minimum DFSR code in D following the DFSR order $<_D$. The minimum DFSR code is a canonical label of I.

To compute DFSR codes, we replace at first each type of node and property in the RDF triple store by an integer *ID* in such way to maintain lexicographical order of string (URI type and URI property). From the RDF triple store of Figure 2 we obtain the following mapping of classes and properties: *age=1*, *city=2*, *hasAddress=3*, *hasFather=4*, *hasfriend=5*, *hasParent=6*, *name=7*, *street=8*, *Lecturer^Researcher=9*, *Male^Person=10*, *Person=11*, *Resource=12*. We assign the code 0 to literals. Figure 4 shows an IRDF graph pattern with instances in Figure 2 and its DFSR

code. To choose the first 5-tuple of the minimum DFSR code we use a lexicographic order on the *IDs* of properties as in (Maduko et al., 2008). Therefore, the first 5-tuple in a minimum DFSR code is the one corresponding to the untyped RDF triple with the minimum property in the IRDF graph pattern. When an IRDF graph pattern has more than one untyped RDF triple with minimum property we use a lexicographic order on the IDs of the subjects first and the objects if needed to choose our first 5-tuple. When an IRDF graph pattern has *n (n>1)* minimum RDF triples (same property, subject and object) we compute *n* DFSR codes and choose the minimum one following DFSR order. By adding a lexicographic test between subjects and between objects we reduce the cases where we have more than one minimum RDF triple and therefore we reduce the number of DFSR codes computed. In Figure 4 the node *Lecturer^Researcher* has the discovery time 1 because *hasFather* is the minimum property following lexicographic order and *Lecturer^Researcher* is the subject of this triple. From *Lecturer^Researcher*, we do a Depth-First Search using lexicographic order on properties, subjects and object to obtain the other discovery times.

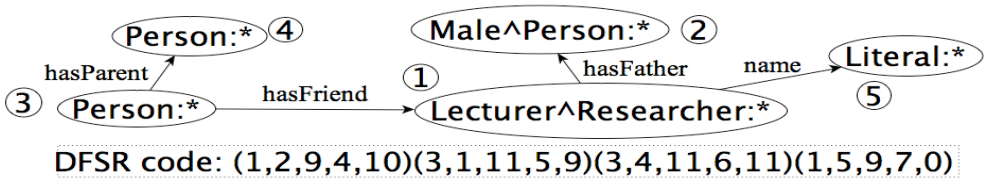


Figure 4. IRDF Graph pattern and its minimum DFSR code.

DETAILED INDUCTION ALGORITHM TO CREATE A FULL INDEX

Definition 11: Kernel of IRDF graph patterns, kernel of DFSR codes. Given *I* and *J* two IRDF graph pattern of size *s >1* sharing *s-1* IRDF triples and *D* (resp. *E*) the minimum DFSR code of *I* (resp. *J*). We call kernel of *I* and *J* the IRDF graph patterns *K* of size *s-1* containing the shared IRDF triples of *I* and *J*. The minimum DFSR code of *K* is the kernel of *D* and *E*.

Definition 12: Specific Infered RDF triples (Specific IRDF triples). Given an IRDF graph patterns *I* and a kernel *K* we call Specific Infered RDF triples of *I* in relation to *K* and note it $I_{\setminus K}$ the IRDF triples in *I* and not in *K*.

Definition 13 : Join IRDF graph patterns. The join of two IRDF graph patterns *G* and *H* of size *s>1* such that *K* is the kernel of *G* and *H*, on their *s-1* shared IRDF triples, is the IRDF graph pattern *J* of size *s +1* such that $J = K \cup G_{\setminus K} \cup H_{\setminus K}$.

To construct our index structure our algorithm relies on these three definitions and the following principles: if an IRDF graph pattern *I* has instances in a RDF triple store then all IRDF graph pattern corresponding to a subset of IRDF triples of *I* has at least as many instances as *I* in the triple store. Level-wise, this gives rise to an efficient construction of DFSR code hierarchy in three phases.

Phase 1: Initialization and enumeration of size 1 DFSR codes

The initialization builds a mapping between each property, type of subject and object with an integer according to the lexicographic order. For instance, *age* in Figure 2 is mapped to 1 and *Person* is mapped to 11. Then to build the first level of the index structure, our algorithm performs a SPARQL query to retrieve all the distinct IRDF graph patterns of size 1 in the triple store. From the list of IRDF graph patterns and the mapping created in the initialization phase, the minimum DFSR codes of size 1 are built. We do not use the kernel notion between levels 1 and 2. A procedure is used to compute the integers corresponding to the property, type of subject and object of each IRDF graph pattern. These integers are the last three elements of the 5-tuple representing the DFSR code (the first and second elements are the discovery times of subject and object). Since we have an IRDF graph pattern of size 1, the discovery time of the subject is 1 and the object one is 2. Algorithm of phase 1 is shown in the following:

```
Procedure DFSROne ()
P: set of IRDF graph patterns of size 1
var level1 = {}
subject, object, property: integer
1. for all untyped RDF triple e in P do
2.      subject= mapping(e.type_subjects) //return the ID corresponding to the type of subject
3.      object = mapping(e.type_objects) //return the ID corresponding to the type of object
4.      property = mapping(e.property) //return the ID corresponding to the property
5.      d=new DFSR(1,2, subject, property, object)
6.      if not(level1.contain(d))
7.              level1 = level1 ∪ {d}
}
```

Algorithm 1. Phase 1: Building the level 1 of the index structure.

Phase 2: Building of size 2 DFSR codes

We fill the second level of the index structure with DFSR codes of size 2 built from DFSR codes of size 1. Algorithm 2 searches for couples of DFSR codes of size 1 which share a node and join hem to obtain a DFSR code of size 2. We distinguish three cases:

Case 1: The 5-tuples of the two joined DFSR codes share an identical subject. In the resulting DFSR code of size 2 the 5-tuple obtained from the minimum DFSR code keep its discovery times (1 for its subject and 2 for its object) and begins the sequence of 5-tuple. The discovery times of 5-tuple arose from the other DFSR code are *(1, 3)*. After building DFSR codes of size 2 we check if the corresponding IRDF graph patterns have at least one instance in the RDF triple store (candidate evaluation phase). Only IRDF graph patterns with at least one instance in the RDF triple store have their minimum DFSR code added in the index structure. For each DFSR code added in the index, our algorithm marks the two DFSR codes joined to obtain it, as they are included in the resulting DFSR code. With this mark we are able at the end of the algorithm to show all IRDF graph pattern in the index or only the IRDF graph pattern with maximal coverage (IRDF graph pattern without mark). Figure 5 shows an instance of DFSR code of size 2 built from 2 DFSR codes of size 1 in case 1.

IRDF graph patterns of size 1 joined to build IRDF graph patterns of size 2 that are kept after the evaluation phase are marked as disposable (they become redundant with the IRDF graph patterns

of size 2) so at the end we are able to show all IRDF graph patterns in the index structure or only the IRDF graph patterns with maximal coverage (the concise index).

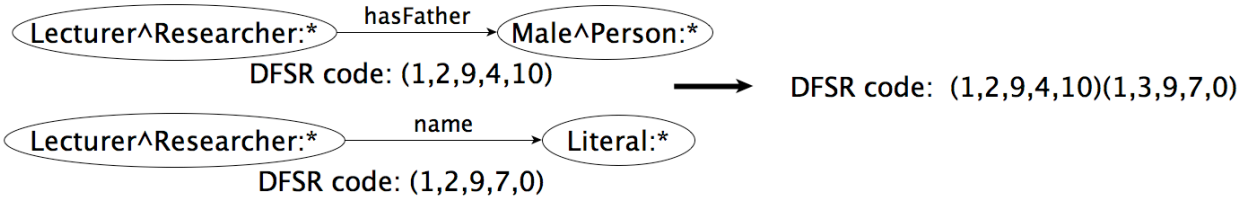


Figure 5.Example of join on two DFSR codes with identical subject.

Case 2: The subject of the 5-tuple of one joined DFSR code is identical to the object of the 5-tuple of the other joined DFSR code. In the resulting DFSR code of size 2 the 5-tuple obtained from the minimum DFSR code keep its discovery times *(1, 2)*. If the 5-tuple of the minimum DFSR code is the one that shares its subject, the 5-tuple of the other DFSR code has for discovery times *(3, 1)*. Otherwise the 5-tuple of the other DFSR code has for discovery times *(2,3)*. The remaining process is similar to the one detailed in case 1.

Case 3: The 5-tuples of the two joined DFSR codes share an identical object. In the resulting DFSR code of size 2 the 5-tuple obtained from the minimum DFSR code keep its discovery times (1,2). The discovery times of 5-tuple arose from the other DFSR code are *(3, 2)*. Building, checking and initializing the DFSR code of size 2 follow the same process as in case 1.

```
Procedure DFSRTWO ()
P: set of DFSR code of size 1
var level2 = {}
1. for all DFSR codes d1 in P do
2.   for all DFSR codes d2 in P do
3.       Case 1: d1.tuple.subject = d2.tuple.subject
4.               d=new DFSR(d1.tuple,d2.tuple,1,3)
5.               if(instanceInRep(d)) then {
6.                       level2 = level2 ∪ d
7.                       marked(d1) //marked
8.                       marked(d2) } //marked
9.       Case 2: d1.tuple.subject = d2.tuple.object
10.              if(d1<d2) then
11.                      d2.setDiscoveries(3,1)
12.              else
13.                      d1.setDiscoveries(2,3)
14.              d=new DFSR(d1,d2)
15.              if(instanceInRep(d)) then {
16.                      level2 = leve2 ∪ d
17.                      marked(d1) //marked
18.                      marked(d2) } //marked
19.      Case 2 bis: d1.tuple.object = d2.tuple.subject
```

```
20.             permute d1 and d2
21.             goto case 2
22.     Case 4: d1.tuple.object = d2.tuple.object
23.             d=new DFSR(d1.tuple,d2.tuple,3,2)
24.             if(instanceInRep(d)) then {
25.                     level2 = level2 ∪ d
26.                     marked(d1) //marked
37.                     marked(d2) } //marked
}
```

Algorithm 2. Phase 2: Building the level 2 of the index structure.

The three previous cases are not disjoint. So, the result of a join operation may be zero to four DFSR codes. During the candidate evaluation phase, DFSR codes are translated into RDF to search if the candidate IRDF graph pattern has at least one instance in the triple store. Therefore, our algorithm constructs from a DFSR code a SPARQL query to search instances of the corresponding IRDF graph pattern.

Phase 3: recursive discovery of DFSR codes of size *n*.

At this step, the join operator is applied on two DFSR codes of size *s-1* (*s>2*) to obtain a DFSR code of size *s*. Our algorithm searches for couples of DFSR codes that share a kernel. Before keeping the DFSR code resulting from the join operation, the algorithm checks (i) if the newly generated IRDF graph pattern is not redundant with an IRDF graph patterns already generated at the current level and (ii) if the IRDF graph pattern has at least one instance in the RDF triple store. The DFSR codes of size *s-1* joined to obtain a kept DFSR code of size *s* are marked as disposable. For example the join operation in Figure 5 is successful (meaning we have at least one instance of *Lecturer^Researcher* with a name and a *father* in the RDF triple store) and the IRDF graph pattern of size 1 *[Lecturer ^Researcher:*]-name->[Literal:*]* and *[Lecturer ^ Researcher:*]-hasParent-> [Male ^ Person:*]* are marked as disposable. So at the end of the process, all unmarked IRDF graph patterns represent the IRDF graph patterns with maximal coverage. The join sub procedure returns one or two DFSR codes. The procedure DFSRN checks for each DFSR code returned if it is not redundant and has at least one instance.

```
Procedure DFSRN (P: set of DFSR code previous level)
var levelN = {} identifier = 0
1. for all DFSR codes d1 in P do
2.    for all DFSR codes d2 in P do
3.       if(kernel(d1,d2)) then {
4.             d = join(d1,d2,kernel)
5.             for each DFSR code di in d
6.                    if(di not in levelN) {
7.                           if(di is frequent) then {
8.                                  levelN = levelN ∪ di
9.                                  marked(d1)
10.                                 marked(d2) }}
```

```
11.                 else { //di already in levelN
12.                         marked(d1)
13.                         marked(d2) } }
}
```
Algorithm 3. Phase 3: Building the level s (s>2) of the index structure.

The detail of the join operator is shown in the following.

SubProcedure join (d_G:DFSR code , d_H : DFSR code, k : kernel of d_G and d_H)
```
1.      eG = specificTupleKernel(dG,k) //dG\k
2.      eH = specificTupleKernel(dH,k) //dH\k
3.      dG = dG - eG ; dH = dH - eH
4.      tG = linkToKernel(dG, eG)
5.      tH = linkToKernel(dH, eH)
6.      tH1 = uniqueTupleThrougth(dG,dH,tG)
7.      if tH1 < 0 then { //no unique time
8.              times = timesPossible()
9.              tH1 = chooseOne(times, dG, dH)
10.     }
11.     tH2=secondTime(eH,eG,tG,tH)
12.     for all time t in tH2 do {
13.             eG.setDiscoveries(tH1,t)
14.             dH.addTuple(eG) ; dH.addTuple(eH)
15.             dH.sort()
16.             res.add(dH)
17.     }
        return res
}
```
Algorithm 4. Join operator for phase 3.

Line 1 and 2 of the join procedure take away from each DFSR code the 5-tuple corresponding to the Specific IRDF triples of G and H in relation to K. In line 3 we keep in d_G and d_H the 5-tuples corresponding to kernel k. We can decompose the remaining lines in three parts:

Part 1 (line 3 to 9). The aim of the first part of the algorithm is to find the discovery time t_{H1} to link one of node of the specific 5-tuple in the first DFSR code d_G to the kernel of the second DFSR code d_H under consideration to generate a new DFSR code of size s. After Line 3 we have in d_G and d_H only the 5-tuples in kernel k. Lines 4 and 5 retrieve the node of e_G (resp. e_H) and its discovery time t_G (resp. t_H) with respect to the kernel in d_G (resp. d_H). To map the discovery time t_G in d_G to a discovery time t_{H1} in d_H we distinguish two cases:

Case 1. First (line 6), we search if there is a tuple in d_G, which has a node with the discovery time t_G and which is unique in the kernel. If it is the case, then the corresponding edge in d_H provides the discovery time t_{H1} corresponding to t_G. This process could be assimilated to the first step of an isomorphism test between two DFSR codes. A complete isomorphism test is done only when we could not find a unique tuple. In most cases, only this first case is required

and the algorithm does not need isomorphism test. Figure 6 shows an example of such a computation.

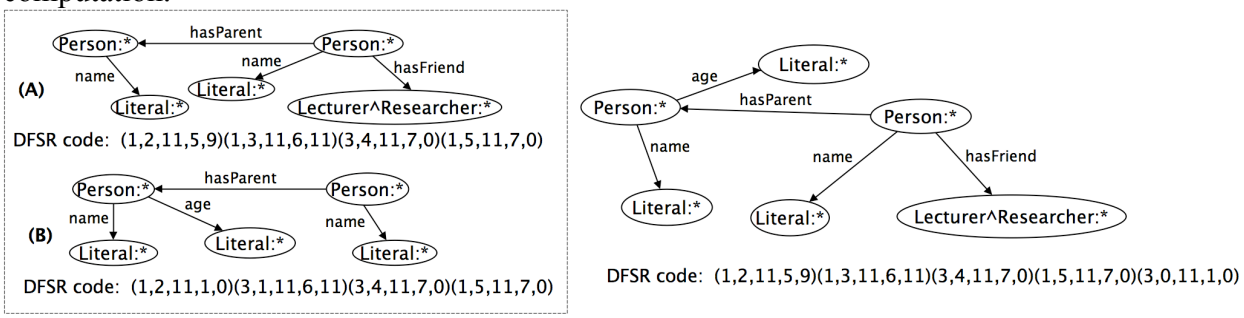


Figure 6. Example of joining two IRDF graph patterns of size 2, case 1.

Case 2. If the first case fails the sub-procedure timePossibles() returns all the candidate discovery times. In our algorithm the sub-procedure chooseOne returns the time t_{H1} in d_H corresponding to t_G in d_G. To do that, each candidate discovery time is compared to t_G by searching every path going through the node corresponding to t_G. Figure 7 shows an example of such a case.

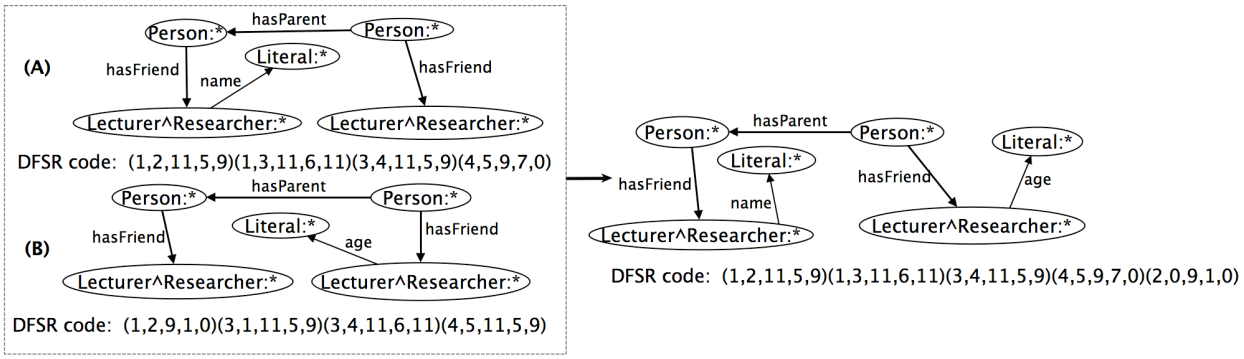


Figure 7. Example of joining two IRDF graph patterns of size 4, case 2.

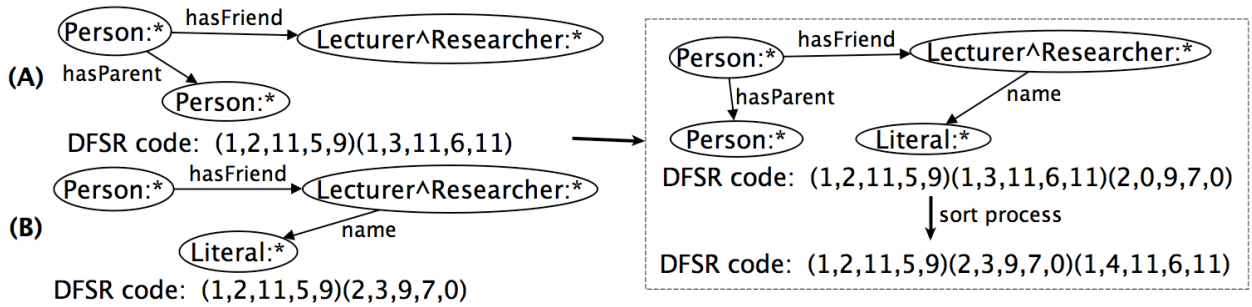


Figure 8. Join two IRDF graph patterns, when the two unlinked nodes of the specific IRDF triples are different.

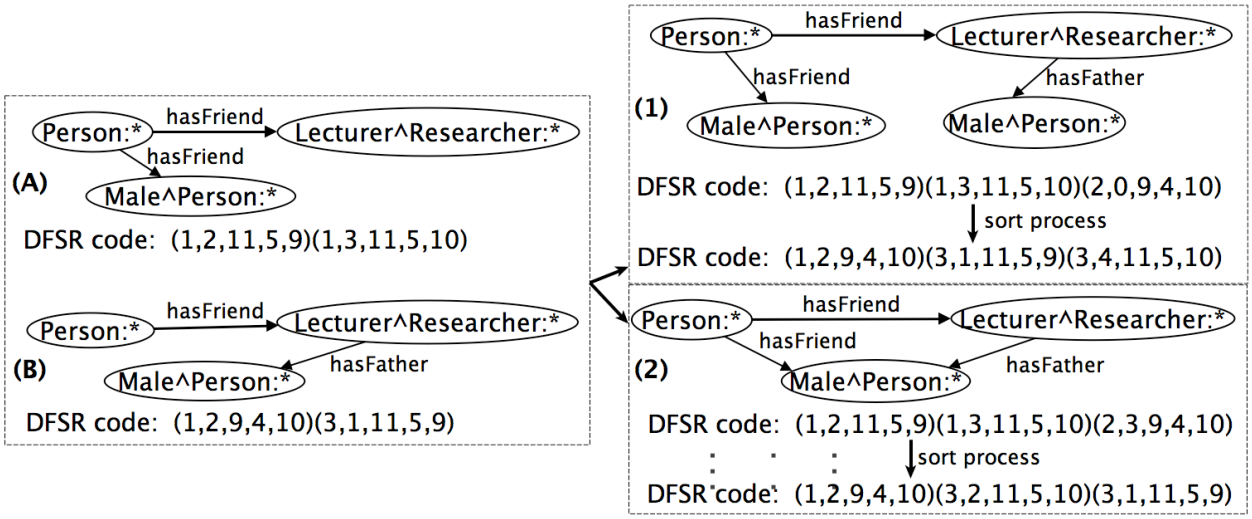


Figure 9. Join two IRDF graph patterns when the two unlinked nodes of the specific IRDF triples are identical.

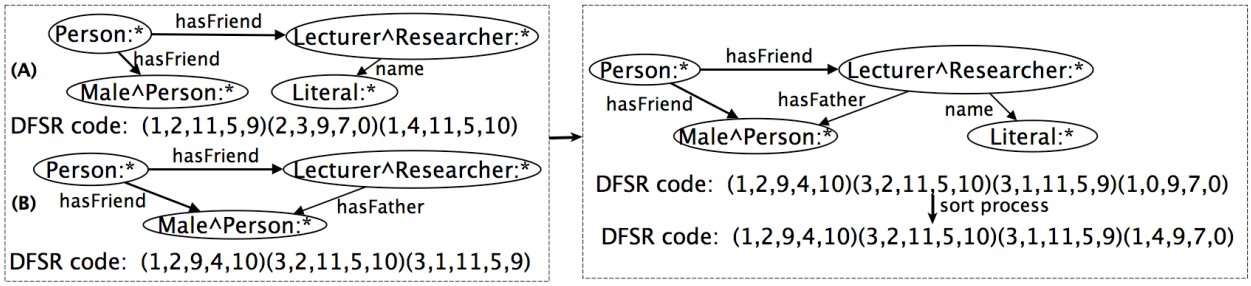


Figure 10. Join two IRDF graph patterns with one of specific IRDF triples linked to kernel by two nodes.

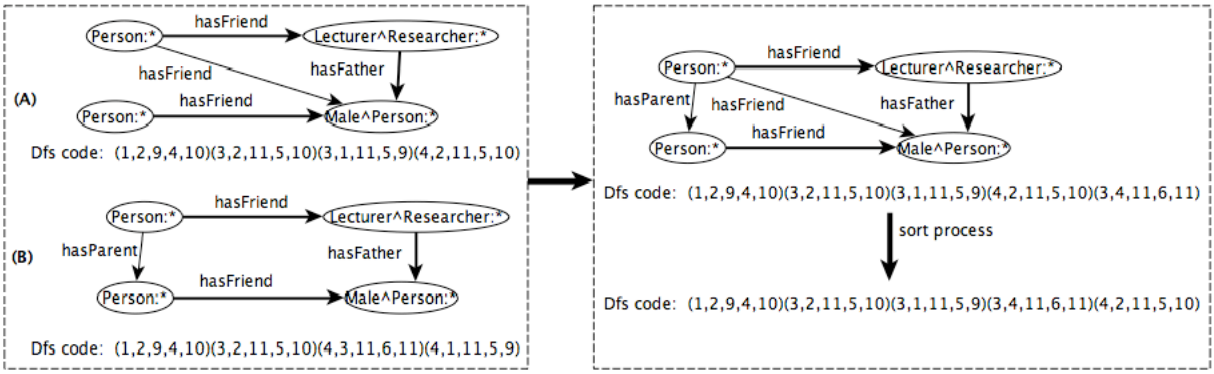


Figure 11. Join two IRDF graph patterns with the two specific IRDF triples linked to kernel by two nodes.

Part 2 (line 11). After finding the discovery time corresponding to t_G, the next step is to retrieve the discovery time t_{H2} in d_H corresponding to the discovery time of the other node of e_G in d_G. We have three cases:

Case 1. The two specific tuples are linked to the kernel by one node, with two subcases:

Case 1.1. The two unlinked nodes of the specific edges are different or both typed as literal. In the only resulting IRDF graph pattern, the two nodes remain unlinked. We set to 0 the discovery time t_{H2} in d_H corresponding to the discovery time of the other node of e_G in d_G. The final sorting process resets all the discovery times following a Depth-First-Search. Figure 8 shows an example of such a case.

Case 1.2. The two unlinked nodes of the specific edges are identical and not literals. In this case we obtain two IRDF graph patterns. The first (Figure 9 (1)) is obtained following the case a) with $t_{H2}=0$. The second IRDF graph pattern (Figure 9 (2)) is obtained by merging the two unlinked nodes of the specific 5-tuple. So the discovery time t_{H2} in d_H corresponding to the discovery time of the other specific node of e_G in d_G is set to the discovery time of the unlinked node of e_H in d_H. Figure 9 shows such a case.

Case 2. Only one of the specific tuple is linked to the kernel by its two nodes. The second specific edge is linked to the kernel by one node. To optimize, we choose to add the second specific edge to the first IRDF graph pattern thus we do not need to compute the search a second time. In this case we obtain one IRDF graph pattern. Like in case *1.1* the unlinked node of the second specific edge has its discovery time set to *0*. Figure 10 shows an example of such a computation.

Case 3. The two specific tuples are linked to the kernel by their two nodes. In the unique IRDF graph pattern resulting from the join, the two specific tuples are still linked to the kernel with their two nodes. To retrieve the time t_{H2} in d_H corresponding to the discovery time of the other specific node of e_G in d_G we rely on the process used to find the discovery time of the first linked node. Figure 11 shows an example.

Part 3 (line 12 to 18). After computing the two discovery times t_{H1} and t_{H2}, lines 12 to 18 add the 5-tuple e_G and e_H to DFSR code d_H. At first we reset the discovery times of e_G with t_{H1} and t_{H2}. Then the 5-tuples e_G and e_H are added in d_H that is sorted. As in phase 2, IRDF graph patterns of size s-*1* joined to build IRDF graph patterns of size s that are kept after the evaluation phase are marked as disposable.

Finally, note that we combine in our algorithm like (Yan & Han, 2002) the growing and checking of DFSR code into one procedure, thus accelerating the mining process. Our algorithm stops at level s if there is no IRDF graph pattern of size s with at least one instance in the RDF triple store.

At the end of this process the DFSR codes kept in the index are translated in RDF graphs and can be loaded in a dedicated named graph of the triple store allowing anyone to query the index in SPARQL. For instance deciding whether a query can find answers in a store amounts to solving an ASK with the pattern of that query on the named graph containing the index.

EXPERIMENTS AND PERFORMANCES

Our algorithm relies on the CORESE/KGRAM (Corby et al., 2004 ;Corby, 2008) implementing SPARQL 1.0 and SPARQL 1.1 recommendations with some minor modifications and some extensions. The building of the index of an RDF triple store is done after all the inferences (mostly the RDFS entailments) have been done and the dataset has been enriched with the

derivations they produced. More details on the formal semantics of the underlying graph models and projection operator are available in (Baget et al., 2008). Our algorithm is designed for endpoints publishing data. If an endpoint restricts accesses the algorithm is run on the public part for the public index and/or the full index being part of the base will be subject to access control like the rest of the base. In this paper, we focused on generating indexes and considered access control out of scope. We tested our algorithm on a merge of three datasets: personData of DBPedia exhibiting IRDF graph patterns in form of stars (Gandon, 2003; Gandon et al., 2008); a foaf dataset used by our team in teaching semantic Web and ensuring the presence of blank nodes, multi typed nodes and cycles; and a tag dataset extracted from delicious with paths and stars combined as structure of IRDF graph patterns. The resulting dataset contains 149,882 triples and includes arbitrary IRDF graph patterns. We obtain the results shown in Figure 12: *RP* represents the number of IRDF graph patterns not kept because redundant with other IRDF graph patterns of the same size. *MP* is the number of IRDF graph patterns marked as disposable. *UP* is the number of unmarked IRDF graph patterns in the final index structure and *JO* is the number of join operations done. The sum of *MP* and *UP* is the number of IRDF graph patterns in the index structure.
Figure 13 shows the number of IRDF graph patterns not kept because they had no instances in the RDF triple store (*NF*) and Figure 14 shows the computation times (*CT*) in second per level.

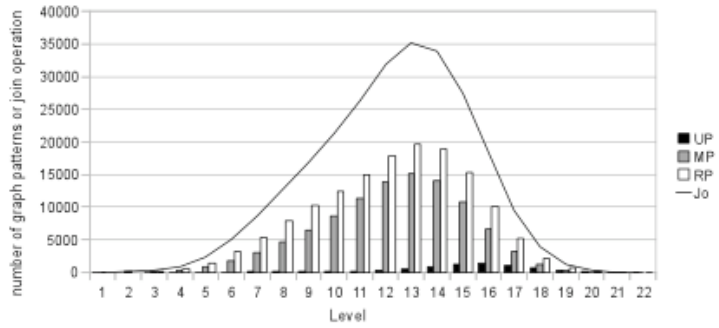


Figure 12. Number of IRDF graph patterns, redundancies and join operation used by level.

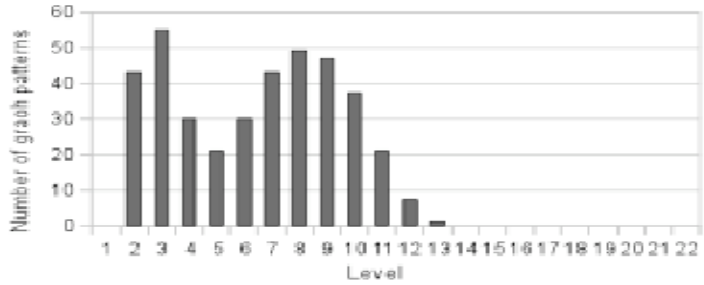


Figure 13. Number of IRDF graph patterns not kept.

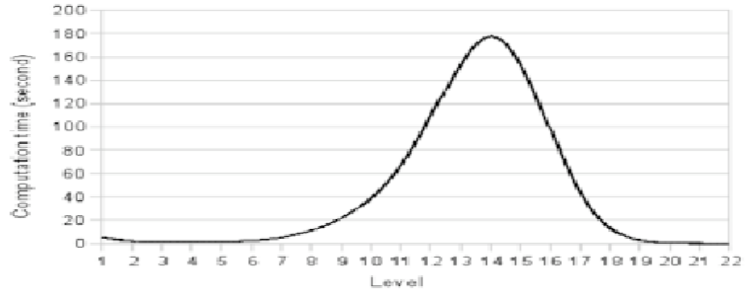


Figure 14. Computation time per level.

We can distinguish 4 periods in our computation:

- Level 1: The computation time of the SPARQL query at level 1 depends strongly on the size of the triple store. In our case the computation time of level 1 is greater than computation time of levels 2 to 6 due to the large size of our triple store in relation to the low number of join operation computed at levels 2 to 6.
- Level 2 to 6: The computation time is near zero due to the low number of join operation computed.
- Level 7 to 13: The number of IRDF graph patterns in the index structure and so the number of join operations increases quickly involving a high number of triple store access and isomorphism tests, increasing the computation time.

Level 14 to 22: *NF* and *RP* are the two ways used to avoid an overflow of IRDF graph patterns. From level 14 all the candidate IRDF graph patterns have at least one instance in the RDF triple store (*NF=0* in Figure 13). The number of IRDF graph patterns generated decreases and is equal to 1 at level 22. From level 14 the algorithm converges quickly and the computation time is near zero from level 19.

The percentage of IRDF graph patterns marked disposable is *90.44%.* The number of redundancies between join operations is high and one of our perspectives is to reduce it.

Complexity Analysis

In graph query processing, the complexity time of level 1 is $O(na) + Tq$ where na is the number of triples in the RDF store and Tq is the time to compute the SPARQL query of level 1. Tq depends also to the number of triples in the RDF triple store. The complexity time of level 2 is $O(nb_1^2)(Tq_2 + O(ni_2))$ where nb_i is the number of IRDF graph patterns of size i ($i > 0$), ni_2 is the average number of instances of IRDF graph pattern of size 2 and Tq_2 is the average time to compute a SPARQL query finding instances of an IRDF graph pattern of size 2. The complexity time of level $s>2$ is $O(nb_{s-1}^2[s^s + Tq_s + ni_s])$ where s is the level, Tq_s is the time to compute the SPARQL query finding instances of an IRDF graph pattern of size s, ni_s is the average number of instances of IRDF graph pattern of size s in the triple store.

INCREMENTAL CHANGES OF THE INDEX WHEN UPDATING THE STORE

Insertion or deletion of annotations in the triple store may cause changes to the index and it is important to have an incremental algorithm to update it. We use listeners on the triple store to register for events we are interested to update the index.

Insertion of a*nnotation*s in the triple store

When an annotation, that represented by a RDF graph containing several triples, is inserted, we distinguish four phases to update the index structure: (1) initialization, (2) update of level 1, (3) update of level 2, (4) update of level $n>2$.

Phase 1 - Initialization phase. We perform a SPARQL query to retrieve the IRDF graph patterns of size 1 from the given annotation and build their corresponding DFSR codes using the procedure *DFSROne* (algorithm 1). From the list of DFSR codes of size 1 obtained in this phase the algorithm updates the index structure.

Phase 2 - update of level 1. We check if each DFSR code in the list obtained in the initialization phase is already or not in the level 1 of the index structure. If a DFSR code is not in the index structure it is inserted at the level 1. If a DFSR code is already in the index structure it makes no change to the level 1 of the index structure. In any case we keep the corresponding DFSR code in the list of inserted codes that may cause insertions at level 2 of the index structure.

Phase 3 - update of level 2. The DFSR codes, in the list of inserted DFSR codes at level 1, are joined with the other DFSR codes at level 1 which 5-tuple share at least one node following algorithm 2. The resulting DFSR codes are classified in three categories:

Category 1: The resulting DFSR code is already in the index and was generated using the same join operation (same DFSR codes joined). This DFSR code makes no change to the level 2.

Category 2: The resulting DFSR code is already in the index structure and was generated using another join operation. The DFSR code is inserted in level 2 and marked disposable.

Category 3: The resulting DFSR code is not in the index structure. The new DFSR code is inserted in the level 2.

At the end of each category we add this DFSR code in the list of inserted code because it may cause insertions at level 3.

Phase 4: update of level *n* (*n>2*). The DFSR codes in the list of inserted codes at level *n-1* are joined with the other DFSR codes at level *n-1* which share a kernel following algorithm 3. The resulting DFSR codes are classified in three categories as in phase 3 and their identifiers are added in the list of inserted codes at level *n*.

Our algorithm stops at level *s* (*s >1*) if there is no identifier of DFSR code in the list of inserted codes at level *s-1*. It means that we have no DFSR code in level *s* generated from a DFSR code in the list built at level *s-1*.

Complexity Analysis. The complexity time of the update of level 1 is $O(nb_1)$ where nb_1 is the number of IRDF graph patterns of size i ($i > 0$). The complexity time of level 2 is $O(nb_1 * nb_2 + nb_1(Tq_2 + ni_2))$ where ni_2 is the average number of instances of IRDF graph pattern of size 2 and Tq_2 is the average time to compute a SPARQL query finding instances of an IRDF graph pattern of size 2. The complexity time of level $s>2$ is $O(nb_{s-1}(nb_{maj}(s^s + nb_s + Tq_s + ni_s)))$ where s is the level, nb_{maj} is the number of IRDF graph patterns updated in level $s-1$, Tq_s is the time to compute the SPARQL query finding instances of an IRDF graph pattern of size s, ni_s is the average number of instances of IRDF graph pattern of size s in the triple store.

Deletion of annotations in the triple store

When an annotation is deleted from the triple store we distinguish 3 phases to update the index structure: (1) initialization, (2) update of level 1, (3) update of level $n>1$.

Phase 1 - Initialization phase. The process is identical with the initialization phase in case of insertion of annotation. At the end of this phase we obtain a list of DFSR code of size 1 corresponding to the IRDF graph patterns build from the deleted annotation.

Phase 2 - update of level 1. For each DFSR code in the list obtained in initialization phase, we check with a SPARQL query if it has still some instances in the triple store to keep it in the index. We distinguish two cases:

Case 1: An IRDF graph pattern has still instances in the triple store. We add the corresponding DFSR code obtained in phase 1 in a list named *checkList* because it may cause deletions in level 2.

Case 2: An IRDF graph pattern has no instance in the triple store. The DFSR code obtained in phase 1 is deleted from the level 1 of the index structure and it is added in a list named *delList* because all the DFSR codes of level 2 generated from it have to be deleted.

Phase 3: update of level *n* (*n>1*). We iterate on the DFSR codes previously inserted in *delList* and *checkList*.

Iteration on delList codes. Each DFSR code of level n generated from a DFSR code which is in the *delList* list of level $n-1$ is deleted from the index structure and we add it in the *delList* list of level n.

Iteration on checkList codes. Each DFSR code of level n generated from a DFSR code which is in the *checkList* list of level $n-1$ is checked in the triple store. We distinguish two cases:

Case 1. There is no instance of the IRDF graph pattern corresponding to the DFSR code in the triple store. The DFSR code is deleted from the index structure and we add it to *delList* at level n.

Case 2. There is at least one instance of the IRDF graph pattern corresponding to the DFSR code in the triple store. We add the DFSR code to the *checklist* of level n.

To update the level $n+1$ the lists *checklist* and *delList* generated at level n are used.

The algorithm stops when the higher level of the index has been updated or when the lists *checkList* and *delList* at level $n-1$ are empty.

Complexity Analysis. The complexity time of the update of level 1 is $O(nb_1 * Tq_1)$ where nb_i is the number of IRDF graph patterns of size i ($i > 0$) and Tq_i is the average time to compute a SPARQL query finding instances of an IRDF graph pattern of size $i > 0$. The complexity time of level $s > 1$ is $O(nbs^2(nbd_{s-1}+nbv_{s-1})+ nbd_{s-1}*nbv_{s-1}*Tq_s)$ where nbd_{s-1} and nbv_{s-1} are respectively the number of DFSR codes in *delList* and *checkList* of level $s-1$.

RELATED WORK

A usual representation of an index structure is a hierarchy organized into different levels according to the size of the indexed items. In the literature, approaches differ with regards to the structure of the indexed items. By extending the join index structure studied in relational and spatial databases, (Han et al., 1999) proposed, as basic indexing structure: pairs of identifiers of objects of two classes that are connected via direct or indirect logical relationships. (Stuckenschmidt et al., 2004) extended this approach to propose an index built as a hierarchy of paths. (Vanetik et al., 2002; Yan & Han, 2002) proposed a hierarchical index structure including both path-patterns and star-patterns. (Yan et al., 2004) showed some disadvantages of path-based approaches, and in particular that part of the structural information is lost and that the set of paths in a dataset is usually huge. To overcome these difficulties, (Yan et al., 2004) proposed to use frequent subgraph patterns as basic structures of index items since a graph-based index can significantly improve query performance over a path-based one. The approaches to frequent graph pattern discovery iterate mainly on two phases: the generation of candidate patterns and the evaluation of candidate patterns. The key computational issues are (i) managing and processing redundancies (this problem is particularly challenging due to the NP-hard subgraph isomorphism test), (ii) reducing the size of the index structure and (iii) proposing a join operator to compute efficiently a graph pattern of size s from two graph patterns of size $s-1$ sharing $s-2$ edges. Among the different algorithms we distinguish mainly two approaches to deal with redundancies: *(1) Algorithms using a canonical form to efficiently compare two graph representations and rapidly prune the redundancies in the set of generated candidates*. (Inokuchi et al., 2000) uses an adjacency matrix to represent a graph, defines a canonical form for normal forms of adjacency and proposes an efficient method to index each normal form with its canonical form. (Han et al., 2007; Maduko et al., 2008; Yan & Han, 2002; Yan et al., 2004) rely on a tree representation which is more compact than an adjacency matrix and maps each graph to a unique minimum DFS code as its canonical label. To discover frequent graph-patterns, (Vanetik et al., 2002) builds candidate graph patterns using frequent paths and a matrix that represents the graph with nodes as rows and paths as columns. (Vanetik et al., 2002) uses a canonical representation of paths and path sequences and defines a lexicographical ordering over path pairs, using node labels and degrees of nodes within paths. *(2) Other algorithms propose a join operator such that every distinct graph pattern is generated only once.* Indeed the major concerns with the join operation are that a single join may produce multiple candidates and that a candidate may be redundantly proposed by many join operations (Kuramochi & Karypis, 2001). (Huan et al., 2003) introduces a join operation such that at most two graphs are generated from a single join operation. The FFSM-Join of (Huan et al., 2003) completely removes the redundancy after sorting the graphs by their canonical forms that are a sequence of lower triangular entries of a matrix representing the subgraph. To reduce the database accesses some approaches like (Kuramochi & Karpis, 2001) use the monotony of the frequency condition to eliminate some candidates.

As several approaches of frequent subgraph discovery, ((Huan et al., 2003; Inokuchi et al., 2000; Kuramochi & Karypis, 2001; Yan & Han, 2002) for instance) we generate candidate graph patterns of size *s* by joining two patterns of size *s-1*. To avoid joining each pair of patterns we add information in each DFSR code to know exactly which pairs of patterns share a kernel and thus can be joined.
In the candidate evaluation phase, most of the algorithms ((Huan et al., 2003; Inokuchi et al., 2000; Kuramochi & Karypis, 2001) for instance) compute the frequencies of candidates with respect to the database content and all frequent subgraph patterns are kept in the index structure. Our index is a hierarchy, not a partition as proposed by (Dolby et al., 2008; Fokoue et al., 2006), and it is designed to be used by machines and humans to understand the content that can be found in a triple store. Their partition may look like the roots of our index but the canonical form defined in (Fokoue et al., 2006) is not equivalent to ours. We also collapse multiple instantiations into conjunctive types wherever possible. In addition, our approach does not require a DL reasoner, does not require the schemas for coherence checking and does not limit to conjunctive queries.
When a class or a relationship between two classes is updated (Han et al., 1999) proposes an incremental update propagation of their partial and complete join index hierarchy starting at the level 1. The update affects only one base join indice (basic element of the index) at level 1 and may affect some join indices at higher levels exactly determined by (Han et al., 1999). The base join index hierarchy (Han et al., 1999) is updated using only the first step of the algorithm proposed for partial and complete join index hierarchy. To handle insertion or deletion of graphs in a graph database, (Yan et al., 2004) proposes an index maintenance algorithm. To update the index (Yan et al., 2004) simply update for each involved fragment its list of graphs containing this fragment. (Yan et al., 2004) notes the quality of the index degrades after a lot of insertions and deletions and propose recomputing the index from scratch. We proposed to use patterns as basic structures of index items like (Yan et al., 2004) but extended to directed labelled multigraph data structure in RDF. To eliminate the redundancies in producing the patterns our algorithm combines the two above-cited alternative solutions during the candidate generation phase: (1) we use trees to represent IRDF graph patterns and a DFSR coding to efficiently compare two IRDF graph patterns and to eliminate redundancies. Then we propose a join operator on two DFSR codes to generate at most four different DFSR codes. Our DFSR coding already extending (Maduko et al., 2008) consists in identifying exactly how edges must be linked during the join operation. In some cases the identification of join point is costly in term of CPU time because it is similar to an isomorphism test. (2) in the candidate evaluation phase the triple store is accessed to eliminate patterns without instance. Note that the pruning step in the candidate evaluation phase is not necessary in our case because our join operator generates only patterns which respect the monotony of the frequency. Note also that we improve the algorithm we proposed in (Basse et al., 2010) to address cyclic graphs, blank nodes and multityped resources. We also added incremental algorithms to update the index when the content of the store changes.

CONCLUSION

In this paper, we presented incremental algorithms to extract and maintain a compact representation of the content of a triple store. We proposed a new DFS coding for RDF graphs, we provided a join operator to significantly reduce the number of generated patterns and we gave the possibility to reduce the index size by keeping only the graph patterns with maximal coverage. In this paper, the motivating scenario was the case of applications exploiting distributed triple stores and justifying the needs for indexes in order to allow humans and machines to know what kinds of knowledge contributions they can expect from a source. The problem of decomposing a query and routing the sub-queries using these indexes remains a research challenge in itself and a perspective for future work.

REFERENCE

Baget, J.-françois, Corby, O., Dieng-kuntz, R., Faron, C., Gandon, F., Giboin, A., Gutierrez, A., et al. (2008). Griwes : Generic Model and Preliminary Specifications for a Graph-Based Knowledge Representation Toolkit. *Structure*, 297-310. Springer.

Basse, A., Gandon, F., Mirbel, I., & Lo, M. (2010). DFS-based frequent graph pattern extraction to characterize the content of RDF Triple Stores. *Proceedings of the WebSci10 Extending the Frontiers of Society OnLine.*

Battle, R., & Benson, E. (2008). Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST). *Web Semantics Science Services and Agents on the World Wide Web*, *6*(1), 61-69.

Corby, O., Dieng-Kuntz, R., & Faron-Zucker, C. (2004). Querying the Semantic Web with Corese Search Engine. *ECAI* (Vol. 16, pp. 1-5).

Corby, O. (2008). Web , Graphs and Semantics. (P. W. Eklund & O. Haemmerle, Eds.)*History*, 5113, 43-61. Springer.

Dolby, J., Fokoue, A., Kalyanpur, A., Ma, L., Schonberg, E., Srinivas, K., & Sun, X. (2008). Scalable Grounded Conjunctive Query Evaluation over Large and Expressive Knowledge Bases. *The Semantic Web ISWC 2008*, 5318(x), 403-418. Springer.

Fokoue, A., Kershenbaum, A., Ma, L., Schonberg, E., & Srinivas, K. (2006). The summary abox: Cutting ontologies down to size. *Lecture Notes in Computer Science*, 4273, 343–356. Springer.

Gandon, F. (2003). Agents handling annotation distribution in a corporate semantic web. *Web Intelligence and Agent Systems.*

Gandon, F., Niang, C., & Lo, M. (2008). Un modèle d'index pour la résolution distribuée de requêtes sur un nombre restreint de bases d'annotation RDF. *Actes dIC* (pp. 25-35).

Han, J., Xie, Z. A., & Fu, Y. (1999). Join Index Hierarchy: An Indexing Structure for Efficient Navigation in Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering TKDE,* 11(2), 321-337.

Han, S., Ng, W. K., & Yu, Y. (2007). FSP: Frequent Substructure Pattern mining. *Proc 6th International Conference on Information Communications Signal Processing* (pp. 1-5).

Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.-U., & Umbrich, J. (2010). Data summaries for on-demand queries over linked data. *In M. Rappa, P. Jones, J. Proceedings of the 19th international conference on World wide web WWW 10* (p. 411). ACM Press.

Huan, J., Wang, W., & Prins, J. (2003). Efficient mining of frequent subgraphs in the presence of isomorphism. *Matrix,* 0, 549. IEEE Computer Society.

Inokuchi, A., Washio, T., & Motoda, H. (2000). An apriori-based algorithm for mining frequent substructures from graph data. *Principles of Data Mining and Knowledge Discovery,* 73(2), 13-23. Springer.

Kuramochi, M., & Karypis, G. (2001). Frequent subgraph discovery. *Proceedings 2001 IEEE International Conference on Data Mining* (Vol. pages, pp. 313-320). IEEE Comput. Soc.

Maduko, A., Anyanwu, K., Sheth, A., & Schliekelman, P. (2008). Graph Summaries for Subgraph Frequency Estimation. *North, 5021,* 508 - 523. Springer-Verlag.

Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., & Broekstra, J. (2004). Index structures and algorithms for querying distributed RDF repositories. *Proceedings of the 13th conference on World Wide Web WWW 04,* 631-639. ACM Press.

Umbrich, J., Hose, K., Karnstedt, M., Harth, A., & Polleres, A. (2011). Comparing data summaries for processing live queries over Linked Data. *World Wide Web Internet And Web Information Systems,* 14(5-6), 1-50. Springer Netherlands.

Vanetik, N., Gudes, E., & Shimony, S. E. (2002). Computing frequent graph patterns from semistructured data. *Proc of 2002 IEEE International Conference on Data Mining ICDM.*

Yan, X., & Han, J. (2002). gSpan: Graph-based substructure pattern mining. *Order A Journal On The Theory Of Ordered Sets And Its Applications,* 02(3), 721-724. IEEE Comput. Soc.

Yan, X., & Han, J. (2003). CloseGraph: mining closed frequent graph patterns. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining* (Vol. 3, p. 286–295). ACM.

Yan, X., Yu, P. S., & Han, J. (2004). Graph indexing: A frequent structure-based approach. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (Vol. ofdatapage, pp. 335-346). ACM.