Dynamic Class Hierarchy Management for Multi-version Ontology-based Personalization

(Article begins on next page)

20 April 2024

Dynamic Class Hierarchy Management
for Multi-version Ontology-based Personalization[1]

Fabio Grandi

Department of Computer Science and Engineering (DISI)
Alma Mater Studiorum – Università di Bologna
Viale Risorgimento, 2
I-40136, Bologna BO - Italy
Tel. +39 051 2093555

`fabio.grandi@unibo.it`

Abstract. We introduce a storage scheme that allows the representation and management of the evolving hierarchical structure of a multi-version ontology in a temporal relational database. The proposed scheme is aimed at supporting ontology-based personalization and temporal access to large collections of resources (data, documents, procedures etc.) stored in a dynamic environment. Whereas in previous works we considered tree-shaped ontologies only, in this work we consider ontologies with a class hierarchy structured as a general directed graph, that is also supporting multiple inheritance and intersection classes. We will also show how multi-version ontologies must be dealt with for the processing of ontology-based personalization queries.

1   INTRODUCTION

In the age of big data, when large amounts of potentially interesting and useful resources are published online day by day, the availability of semantics-aware search engines and intelligent personalization services becomes a key factor for a fruitful exploitation of such resources. In particular, the adoption of reference ontologies and

---

[1]The published version can be found with DOI: 10.1016/j.jcss.2015.06.001

their deployment for the personalization of multi-version resources has been recently proposed by several authors in the medical domain [1,2,3,4] and other application fields (e.g., e-Government [5]). The considered resources range from descriptive data to textual documents, from Web pages to the specification of processes and services. References to ontology classes are added to the computer encoding of resources (e.g., for which an XML [6] format can conveniently be used) to introduce a sort of semantic indexing of contents representing their *applicability*, *relevance* or *eligibility* with respect to ontology classes. Hence, starting from a user-supplied list of ontology classes, a suitable query engine can exploit semantic indexing to retrieve the relevant contents only and produce a *personalized version* of the desired resources.

However, in a dynamic environment, the management of this kind of semantic versioning is interleaved with temporal aspects. For example, we can choose as resources clinical guidelines [7], that is "best practices" encoding and standardizing health care procedures, in textual or executable format, and consider their personalization with respect to an ontology of diseases, patients or available hospital facilities they are applicable to [1]. Personalization will then produce guideline versions tailored to a specific use case. The fast evolution of medical knowledge and the dynamics involved in clinical practice imply the coexistence of multiple temporal versions of the clinical guidelines stored in a repository, which are continually subject to amendments and modifications. Therefore, it is crucial to reconstruct —borrowing the term from the legal domain— the *consolidated version* of a guideline as produced by the application of all the modifications it underwent so far, that is the form in which it currently belongs to the state-of-the-art of clinical practice and, thus, must be applied to patients today. However, also past versions are still important, not only for historical reasons: for example, a physician might be called upon to justify his/her actions for a given patient at a past time on the basis of the clinical guideline versions applicable to the pathology of patient and which were valid at that time (as well as, in the legal domain, a Court might be called to judge today on a crime committed several years ago and for which the normative framework that was in force then has to be applied).

Moreover, in a dynamic environment, the definition of domain ontologies themselves is also subject to modification and, thus, ontologies come out versioned as a consequence of updates periodically effected by domain experts and knowledge engi-

neers, or even by standardization committees. As we showed in [8] for the legal domain (but it also happens for the medical field), personalization of a resource with respect to a past point in time must be effected by taking into account, in order to consider semantic indexing of the desired temporal version of the resource, the version of the reference ontology that was valid at the same time point. In other words, the selected resource version and the ontology version used for personalization must be *mutually temporally consistent*, in order to reconstruct the exact framework in which the resource had been utilized. Since clinical guidelines have also been recently proposed to be used as evidence of the legal standard of care in medical malpractice litigation [9], enforcement of temporal consistency is crucial to assess *a posteriori* the responsibility of physicians having followed the guidelines in the past.

Therefore, in this work we will show how temporal multi-version ontologies can be represented and maintained in a relational setting and how they can be used during the processing of a personalization query. The rest of the paper is organized as follows: in Sec. 2, the ontology-based personalization method proposed in [1,5] is briefly recalled; in Sec. 3, we present our storage scheme and manipulation primitives for temporal versioning of an ontology class hierarchy which is necessary to support such personalization method. Section 4 is devoted to personalization query processing in the presence of a multi-version ontology. Related work is discussed in Sec. 5 and conclusions can finally be found in Sec. 6.

2   A FRAMEWORK FOR ONTOLOGY-BASED PERSONALIZATION

The personalization method proposed in [1,5] is based on the adoption of reference domain ontologies and the introduction of semantic indexing of resource contents with respect to ontology classes. For example, in the medical domain, reference ontologies to be used to this purpose can be derived from the ICD-10[2] international classification of diseases or from the UMLS[3] or SNOMED-CT[4] comprehensive biomedi-

---

[2] http://www.who.int/classifications/icd/en/
[3] http://www.nlm.nih.gov/research/umls/
[4] http://www.ihtsdo.org/snomed-ct/

cal and healthcare terminologies. Semantic indexing can then be used by personalization services to adapt generic resources to specific use cases, for example, to derive and enact individual care plans as proposed in [1,2,3]. Notice that, as a consequence of the information flooding we have been experiencing in recent years, personalization becomes a practical necessity when the huge availability of potentially interesting resources tends to be overwhelming.

Full ontology features, including properties, axioms, expressions and individuals, can be exploited in a processing step, which precedes the personalization process, in order to formalize the personalization context as a set of relevant ontology classes that define a specific use case. For example, in the medical domain, during this phase (called *classification phase* in [5]), a suitable reasoning facility can be used to match the medical records of a patient with the qualifying classes in an ontology of diseases. Then, such ontology classes are used as input of the personalization engine, which retrieves the data resources which are applicable, relevant or eligible with respect to such classes. The only ontology feature which is necessary for the considered personalization approach, on which [1,5,8,10,22] and this paper are focused, is the *hierarchy of classes induced by the IS-A relationship* and, thus, we do not consider in this work properties or other ontology features including the presence of instances. We initially follow the simplified assumption made in the application papers [1,5] and in the preliminary version of this work [10] that the class hierarchy underlying the ontology is tree-shaped, that is each node in the class hierarchy (but the root) has a single parent. However, in this work we will finally remove such limiting hypothesis and also consider general ontologies starting from Sec. 3. Owing to the tree structure, nodes can be assigned a preorder and a postorder code, corresponding to the sequence in which nodes are visited during a preorder or postorder traversal of the tree, respectively. Hence, preorder and postorder codes can be used for characterizing the descendants of a node [11,12]:

$$N \text{ is a descendant of } M \text{ iff } M.Pre < N.Pre \text{ and } N.Post < M.Post$$

(1)

with obvious meaning of the used dotted notation. As we will see in Sec. 4, efficient testing of the descendant relationship is a key feature of personalization query processing [1].

Id	Pre	Post
A	1	7
B	2	1
C	3	4
D	4	2
E	5	3
F	6	6
G	7	5
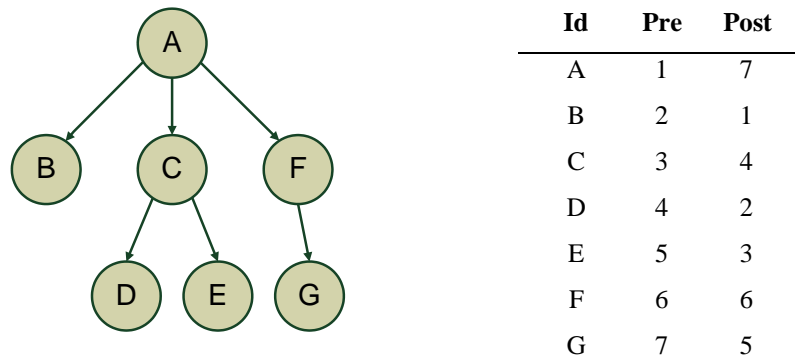
Fig. 1. A sample ontology class hierarchy and its tabular representation

For example, we can consider the sample ontology depicted in the left part of Fig.1, where the corresponding preorder and postorder code of nodes can be found in the table to the right. The structure of the class hierarchy is completely defined by the information present in the table, which, thus, can enable storage of the ontology structure definition in a relational database.

Once defined and stored the ontology structure in this way, node identifiers can be used as a reference to ontology classes for semantic indexing of the resources which are the object of personalization. In [1,5], preorder codes are directly used as node identifiers, whereas as in [10] we will keep them distinguished and associate preorder and postorder codes to time-invariant class identifiers (Id) in order to support ontology versioning. Hence, if the same class belongs to two ontology versions, the class Id is the same in both of them, while the preorder and postorder codes are very likely different as long as the two ontology versions have a different structure. In this way, the proposed encoding scheme implies an *indirect reference* from class identifiers

used for semantic indexing of resources to preorder and postorder codes used for query processing.

In this work, like in [1,5] and [10], we consider personalization of resources with an inner hierarchical organization (e.g., a text organized with chapters, sections, subsections and paragraphs), which, thus, can be easily represented and stored as XML documents [6]. Each XML element in the resource encoding can be represented by means of multiple versions of its contents, each of which can be assigned a temporal validity by means of timestamps [13], and a semantic pertinence by means of references to ontology classes [5].

Owing to the hierarchical organization of resources, temporal validity and semantic applicability properties of an element are inherited by its subelements, unless locally redefined. Considering applicability properties, because of the IS-A semantics (e.g., an individual which is instance of C is also instance of A in Fig.1), if we are looking for all the resource portions that qualify for an instance of an ontology class, we should retrieve the resource portions which are directly applicable to the ontology class itself and also the resource portions which are applicable to its superclasses. For example, if a query retrieves resources concerning an individual belonging to the ontology class E, then the returned resources should be those applicable to class E but also those applicable to the ancestor classes of E (i.e., classes C and A in Fig. 1). Whichever is the most specific class to which our individual of interest belongs, the query results would include all the resource portions applicable to all its ancestor classes up to the ontology root class, which may come out too generic to be of real interest for a specific use case and imply the retrieval of the whole repository, giving up any query selectivity. For instance, considering a repository of resources concerning animals indexed via the "classical" biological taxonomy [5] Domain-Kingdom-Phylum-Class-Order-Family-Genus-Species, this would mean to also retrieve all resources generically applicable to "eukariotes" (i.e., Domain: *Eukarya*), that is the whole repository, when looking for resources concerning "lions" (i.e., Species: *Panthera leo*). In such a case, as proposed in [1,5], using a optional depth parameter in order to focus on the most interesting resources only, the user can limit the applicabil-

---

[5] http://en.wikipedia.org/wiki/Taxonomy_(biology)

ity scope of a query to the ancestors located up to *depth* steps above the most specific class our individual of interest belongs to in the class hierarchy. Hence, we can specify a depth value "0" in order to focus on resources specifically concerning lions only, or a depth value "1" to retrieve resources concerning lions and also other animal species in the lions' superclass (i.e., Genus: *Panthera*).

For example, let us consider the sample chunk in Fig. 2 of a multi-version resource encoded in XML. It is made of an element "foo" with two versions, the former (version 1) valid from T1 to T2 and applicable to class B of the ontology in Fig. 1 and the latter (version 2) valid from T2 on and applicable to class C of the ontology in Fig. 1. The special time value UC (**U**ntil **C**hanged [14]) is used to represent the To value of a right-unlimited time interval. The second version of element "foo" contains a subelement "bar", which inherits the validity of its parent element (i.e., from T2 on) and extends the applicability inherited from its parent also to class G of the ontology in Fig. 1 (i.e., the applicability of "bar" is C or G). The only version (version 1) defined for "bar" is necessary in order to redefine the inherited semantic pertinence (notice that the "pertinence" XML element is defined as a subelement of the "version" XML element).

```
...
<foo>
<version number="1">
  <pertinence>
    <valid from="T1" to="T2"/>
    <applies to="B">
  </pertinence>
      Contents of foo–version 1
</version>
<version number="2">
  <pertinence>
    <valid from="T2" to="UC"/>
    <applies to="C">
  </pertinence>
      Contents of foo–version 2
  <bar>
  <version number="1">
    <pertinence>
      <applies also="G">
    </pertinence>
        Contents of bar–version 1
  </version>
```

```
        </bar>
      </version>
    </foo>
    ...
```

Fig. 2. A chunk of multi-version XML resource.

The XML encoding of multi-version resources exemplified in Fig. 2, which has been proposed in [1,5], has also been adopted in [10] and in this work for the reasons which follow:

- is general enough to be applied to any kind of resources (as it is independent on the non versioned resource schema) and to allow the seamless adoption of an arbitrary number of temporal and semantic versioning dimensions;
- its simplicity allows a self-contained presentation;
- efficient algorithms implemented in a prototype processor are available for personalization query support [1,5] (required extensions to this query engine will be presented in Sec. 4).

However, as far as semantic markup is concerned, other encoding schemes proposed for linking resource contents to ontological information, from the ones proposed as standards like RDFa[6] and microformats[7] to the more exotic ones customary in specific application domains (e.g., biomedical domain), could also be adopted. In such a case, simple modifications, which are beyond the scope of this work, have to be introduced to the personalization query processing methods presented in Sec. 4.

Notice that, in this paper and in [10] we make use of a symbolic example in order to emphasize the generality of our ontology-based personalization approach without sticking to a specific application domain. However, concrete examples of such an approach for personalized access to clinical guidelines in the medical field and to norm texts in the legal field can be found in [1] and [5], respectively.

---

[6] http://rdfa.info/
[7] http://microformats.org/

3 TEMPORAL ONTOLOGY EVOLUTION

In this section, we will introduce primitive operations that can be used for applying structural changes to an ontology and producing a new version, and show how they can be defined in order to maintain a multi-version ontology structure represented and stored as a valid-time relation in a temporal database [13]. In this paper, differently from [10] where we dealt with tree-like ontologies only, we consider general graph ontologies, where each class is allowed to be the child of more than one parent in the class hierarchy, so that intersection classes can be defined and multiple inheritance is allowed. To this aim, we extend our approach presented in [10] to the adoption of the GRIPP numbering scheme used in [15], which provides for the introduction of *non-tree edges* in order to apply the preorder/postorder numbering scheme of trees also to general directed acyclic graphs. The only difference between the numbering scheme adopted here and the original one in [15] is that we assume codes start from 1 (which is, thus, the preorder code of the root node) rather than 0. In practice, we will use a preoder/postorder numbering schemes slightly different from the "classical" scheme used in [11,12], exemplified in Fig. 1, which we also followed in [10] and which was used in the base personalization approach we contributed to in [1,5]. In that case, preorder/postorder codes are incremented only when they are assigned to nodes during the tree visit (as a consequence, they assume all the values in the range range from 1 to the number of nodes N). Indeed, in the GRIPP numbering scheme, visit codes are incremented each time a node is traversed and assigned as preorder/postorder codes to nodes when the traversal is in the right order (hence, maximum preorder/postorder codes are usually greater that N, as all nodes but the leaves are traversed two —once downwards and once upwards— or more times —when they have multiple children— during the visit). For instance, in the ontology structure of Fig. 3, the order in which nodes are visited is A:1 B:2 A:3 C:4 D:5 C:6 E:7 C:8 A:9 F:10 Ē:11 F:12 G:13 F:14 A:15 (Ē denotes the node E reached from the non-tree edge); in practice, Pre and Post codes correspond to the first and last positions, respectively, in which each node can be found in such a sequence. Notice that, in the preliminary work [10], dealing with tree-like ontologies only, we used the "classical" preorder/postorder encoding because it was the numbering scheme adopted in our previous approaches to personalization

and exploited by the implemented prototype [1,5]; therefore, the extension presented in [10] could be applied to such approaches in a straightforward way (the available prototype is currently being upgraded in this direction). Although also the classical numbering scheme can easily be extended to the support of non-tree edge, in this work we preferred to opt for the new numbering scheme, because it gives rise to a bit simpler and more efficient evolution algorithms (e.g., they do not require the evaluation of the maximum preorder code in a subtree as the algorithm in [10; Sec. 3.1.1]); as to the prototype upgrade, the deployment of general graph ontologies in the functionalities of our previous personalization architecture has anyway a big impact on its organization, which also justifies the change of the underlying numbering scheme.

Hence, the table schema used for the representation of an evolving ontology has to be extended with a further attribute, Type, whose admitted values are "T" and "N" to represent a tree or non-tree edge leading to the node identified by Id, respectively. In this way, a general graph can be represented as an underlying spanning tree, containing tree edges only, augmented with non-tree edges.

Id	Pre	Post	Type	Lev
A	1	15	T	1
B	2	2	T	2
C	4	8	T	2
D	5	5	T	3
E	7	7	T	3
F	10	14	T	2
E	11	11	N	3
G	13	13	T	3
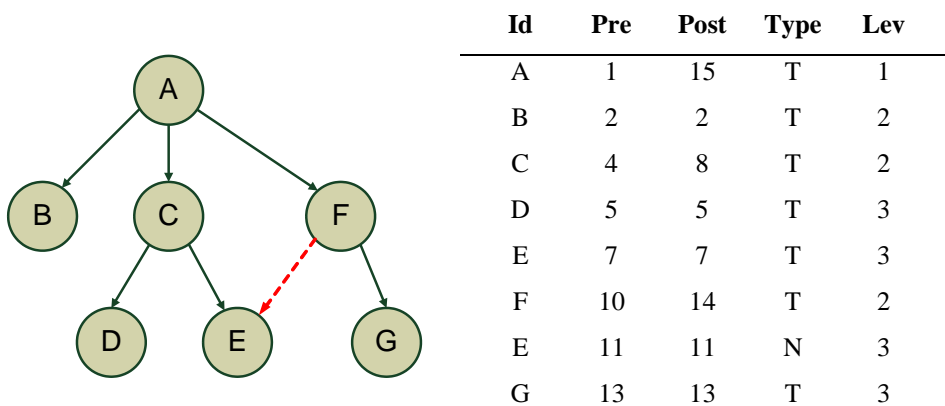
Fig. 3. General graph ontology structure and its tabular representation

The class hierarchy of a sample general ontology and its tabular representation are displayed in Fig. 3, where the non-tree edge has been drawn with a dotted red line. In

particular, the ontology structure displayed in Fig. 3 corresponds to the one introduced in Fig. 1 to which a non-tree edge between class F and class E has been added, so that E becomes a subclass either of class C and of class F.

Furthermore, the table schema has also been augmented by a level attribute, Lev, representing the distance of the node from the top, assuming level 1 for the root node. Actually, level values are not strictly necessary to define the structure of the class hierarchy, but they can conveniently be used for speeding up ontology evolution operations (as described in the subsection that follows) and personalization query processing (as described in Sec. 4).

3.1   General Ontology Structure Evolution Support

In order to support the evolution of an ontology class hierarchy in tabular representation, we introduce five primitive change operations, which can be used in sequence and combination to make arbitrary changes to the ontology structure, and present algorithms to implement their action on the tabular representation exemplified in Fig. 3. For further reference, operations are numbered as **(Oi),** which stands for **O**peration number **i**. Primitives from **(O1)** to **(O3)** represent extensions of the corresponding operations proposed in [10] for three-shaped ontologies, whereas brand new operations **(O4)** and **(O5)** are introduced to manipulate non-tree edges. We assume that all the procedures listed in this subsection work on a table with schema GraphTable(Id, Pre, Post, Type, Lev), corresponding to the tabular representation in Fig. 3, and receive as input parameter a data structure with type GraphRow, that is representing a row of the GraphTable, already containing the Id, Pre, Post, Type and Lev data of the node directly involved by the modification. Starting from the node identifier I, it is straightforward to retrieve such information from the only row with Id="I" and Type="T" in the ontology tabular representation. The same applies to operation **(O4)** involving the creation of a new non-tree edge, for the retrieval of the information concerning the nodes to be connected starting from their identifiers. For operation **(O5)** involving the deletion of an existing non-tree edge, which is defined by the node it connects, the information concerning the edge origin node can still be retrieved from its identifier I in the same way. The information concerning the edge destination

node indeed, starting from its identifier J, can be retrieved from the rows with Id="J" and Type ="N" in the tabular representation with the additional constraint that the destination node is a direct descendant of the origin node (notice that, since any ontology node can receive more than one incoming non-tree edge, more than one row can satisfy the condition Id="J" and Type ="N"). Being M and N the data structures representing the origin and destination nodes, enforcing the additional constraint corresponds to test that condition (1) holds and that N.Lev=M.Lev+1 (i.e., N is a descendant of M directly connected by a single edge).

(O1) Insertion of a leaf node. The operation **InsertUnder(N)** can be used to create a new leaf node as child of the existing node N. If the node N already has children, the new node is created as the rightmost child (the order of siblings does not matter for personalization query processing, as the ancestor-descendant relationships only are relevant). Owing to the definition of preorder, postorder and level codes, the action of the insertion reflects on their values as explained in the following. All the nodes which were visited after N in postorder and N itself must have their postorder code increased by 2 (as they will be visited after the new node and after its parent is traversed upwards). Notice that, being created as the rightmost child of N, the new node will be visited in preorder right after all the nodes in the subtree rooted on N (which satisfy the descendant relations Pre>N.Pre and Post<N.Post). Hence, the nodes which must have their preorder code only increased by 2 are all the nodes which were visited after N both in preorder and in postorder (i.e., nodes visited after N in preorder but not belonging to the subtree rooted on N). The new node must be assigned a preorder and postorder code both equal to the postorder of N plus 1 and a level equal to the one of N plus 1 (type of the new node is obviously "T"). A slightly optimized algorithm for updating the tabular representation is the following:

```
InsertUnder(N:GraphRow)
   NewPP:=N.Post+1;
   ForEach Node in GraphTable Do
      If Node.Post>=N.Post
      Then Node.Post+=2
           If Node.Pre>N.Pre
           Then Node.Pre+=2 EndIf
      EndIf
   EndFor
   AddRow(NewId(),NewPP,NewPP,'T',N.Lev+1)
Return
```

The function NewId() is assumed to create an unused identifier, which acts like a time-invariant key, for the newly added node. Alternatively, the identifier of the new node could be supplied by the user and the procedure has to check that it has not been used yet. The variable NewPP is used to store the value to be eventually assigned both to the preorder and postorder code of the new node, computed before the postorder of N is changed by the rest of the algorithm.

For instance, starting from the ontology in Fig. 3, the execution of the operation **InsertUnder(F)** produces the new ontology version shown in Fig. 4 with the new node created as H.
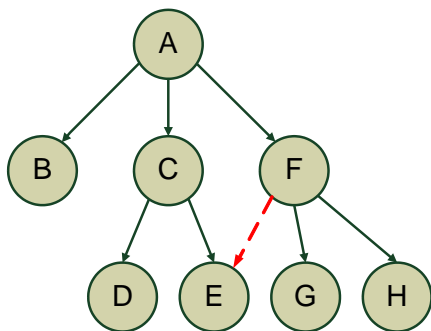
Id	Pre	Post	Type	Lev
A	1	17	T	1
B	2	2	T	2
C	4	8	T	2
D	5	5	T	3
E	7	7	T	3
F	10	16	T	2
E	11	11	N	3
G	13	13	T	3
H	15	15	T	3

(O2) Insertion of an intermediate node. The operation **InsertOver(N)** can be used to create a new node in the path between the node N and its parent (i.e., the new node becomes the new parent of N and a child of the former parent of N). If N is the tree root node, the created node will become the new root. The action of the insertion reflects on the preoder, postorder and level values as explained in the following. All the nodes which were visited after N both in preorder and in postorder must have their preorder and postorder codes increased by 2 (since before they will be reached in both visit orders, the new node will be traversed either downwards and upwards causing a double increment of preorder and postorder codes). All the nodes which were ancestors of N must have their postorder increased by 2 (as they will be reached after the new node is traversed either downwards and upwards causing a double increment of the postorder code). All the nodes which were in the subtree rooted on N (inclusive) must have their preorder, postorder and level increased by 1. The new node inherits the preorder code and the level from N and must be assigned a postorder code equal to the postorder code of N plus 2 (as it is visited after itself is traversed downwards and N is traversed upwards, in addition to the initial postorder of N). A slightly optimized algorithm for accordingly updating the tabular representation is the following:

```
InsertOver(N:GraphRow)
   NewPre:=N.Pre; NewPost:=N.Post+2; NewLev:=N.Lev;
   ForEach Node in GraphTable Do
      If Node.Post>N.Post
      Then Node.Post+=2
           If Node.Pre>N.Pre
           Then Node.Pre+=2 EndIf
      Else If Node.Pre>=N.Pre
           Then Node.Pre++
                Node.Post++
                Node.Lev++
           EndIf
```

```
        EndIf
    EndFor
    AddRow(NewId(),NewPre,NewPost,'T',NewLev)
Return
```

The variables NewPre, NewPost and NewLev are used to compute the codes to be assigned to the new node before the data of N can be changed by the rest of the algorithm.

For instance, starting from the ontology in Fig. 4, the execution of the operation **InsertOver(C)** produces the new ontology version shown in Fig. 5 with the new node created as I.

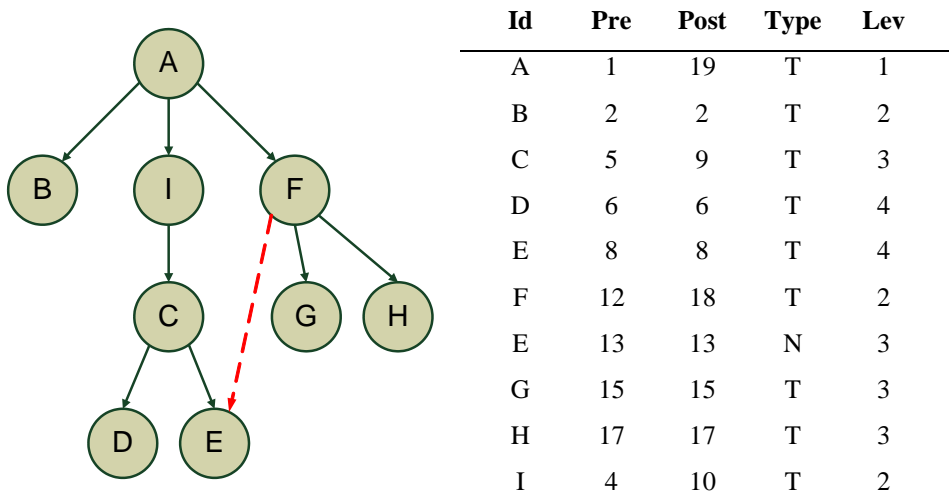| Id	Pre	Post	Type	Lev
A	1	19	T	1
B	2	2	T	2
C	5	9	T	3
D	6	6	T	4
E	8	8	T	4
F	12	18	T	2
E	13	13	N	3
G	15	15	T	3
H	17	17	T	3
I	4	10	T	2

Fig. 5. General graph ontology and its tabular representation

(O3) Deletion of a node. The operation **DeleteNode(N)** can be used to delete node N from the ontology (former children of N become children of the former parent of N). The DeleteNode procedure can be applied to the tree root node only if it has a single

child (which becomes the new root). Moreover, a node cannot be deleted if it has incoming non-tree edges (which, thus, must be removed before the node can be deleted). All the nodes which were visited after N both in preorder and in postorder must have their preorder and postorder codes decreased by 2 (since before they were reached in both visit orders, N were traversed either downwards and upwards causing a double increment of preorder and postorder codes). All the nodes which were in the subtree rooted on N must have their preorder, postorder and level decreased by 1. All the nodes which were ancestors of N must have their postorder decreased by 2 (since before they were reached in postorder, N were traversed either downwards and upwards causing a double increment of the postorder code). An algorithm for updating the tabular representation is the following:

```
DeleteNode(N:GraphRow)
    DeleteRow(N.Id,N.Pre,N.Post,N.Type,N.Lev)
    ForEach Node in GraphTable Do
        If Node.Pre>N.Pre
        Then If Node.Post<N.Post
            Then Node.Pre--
                Node.Post--
                Node.Lev--
            Else If Node.Post>N.Post
                Then Node.Pre-=2
                    Node.Post-=2
                EndIf
            EndIf
        Else If Node.Post>N.Post
            Then Node.Post-=2
        EndIf
    EndFor
Return
```

For instance, starting from the ontology in Fig. 5, the execution of the operation **DeleteNode(B)** produces the new ontology version shown in Fig. 6.
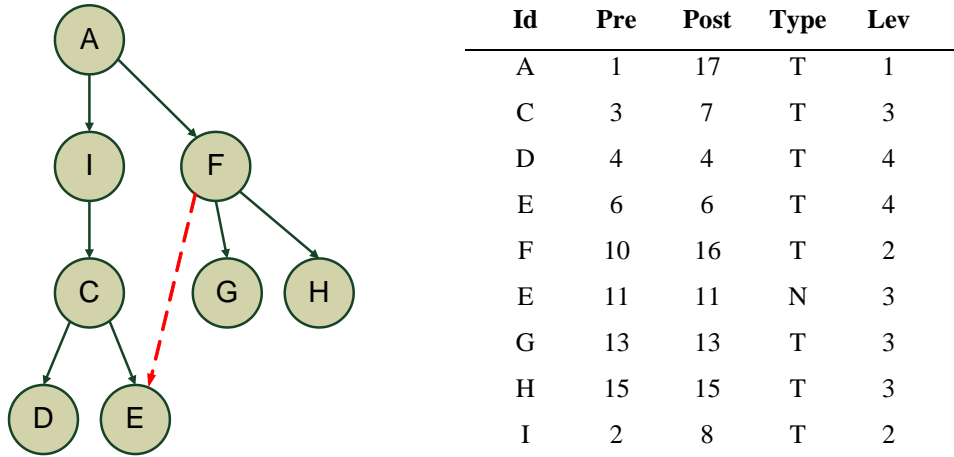
Id	Pre	Post	Type	Lev
A	1	17	T	1
C	3	7	T	3
D	4	4	T	4
E	6	6	T	4
F	10	16	T	2
E	11	11	N	3
G	13	13	T	3
H	15	15	T	3
I	2	8	T	2

Fig. 6. General graph ontology and its tabular representation

(O4) Insertion of a non-tree edge. The operation **InsertEdge(M,N)** can be used to create a new non-tree edge from node M to node N. Before adding it, it must be checked that the two nodes are not already connected and also that the new edge does not close a cycle in the ontology graph. For the last check, verifying that M is not a a descendant of N, either tree and non-tree edges already present have to be considered. All the nodes which were visited after M in postorder and M itself must have their postorder code increased by 2 (as they will be visited, thanks to the new edge, after N and after M is traversed upwards). Node N is visited from M through the new edge from M in preorder right after all the nodes in the subtree rooted on M (which satisfy the descendant relations Pre>M.Pre and Post<M.Post). Hence, the nodes which must have their preorder code only increased by 2 (as they will be visited, thanks to the new edge, after N and after M is traversed upwards) are all the nodes which were visited after M both in preorder and in postorder (i.e., nodes visited after M in preorder but not belonging to the subtree rooted on M). The new tuple representing the new non-tree edge   must be assigned a preorder and postorder code both equal to the postorder of M plus 1 and a level equal to the one of M plus 1 (the assigned type is

obviously "N", to represent that N is reached through a non-tree edge here). A slightly optimized algorithm for updating the tabular representation is the following:

```
InsertEdge(M,N:GraphRow)
   NewPP:=M.Post+1; NewLev:=M.Lev+1;
   ForEach Node in GraphTable Do
      If Node.Post>=M.Post
      Then Node.Post+=2
           If Node.Pre>M.Pre
           Then Node.Pre+=2 EndIf
      EndIf
   EndFor
   AddRow(N.Id,NewPP,NewPP,'N',NewLev)
Return
```

The variables NewPP and NewLev are used to compute the codes to be assigned to the new node before the data of M can be changed by the rest of the algorithm.

For instance, starting from the ontology in Fig. 6, the execution of the operation **InsertEdge(D,G)** produces the new ontology version shown in Fig. 7.

Id	Pre	Post	Type	Lev
A	1	19	T	1
C	3	9	T	3
D	4	6	T	4
E	8	8	T	4
F	12	18	T	2
E	13	13	N	3
G	15	15	T	3
H	17	17	T	3
I	2	10	T	2
G	5	5	N	5
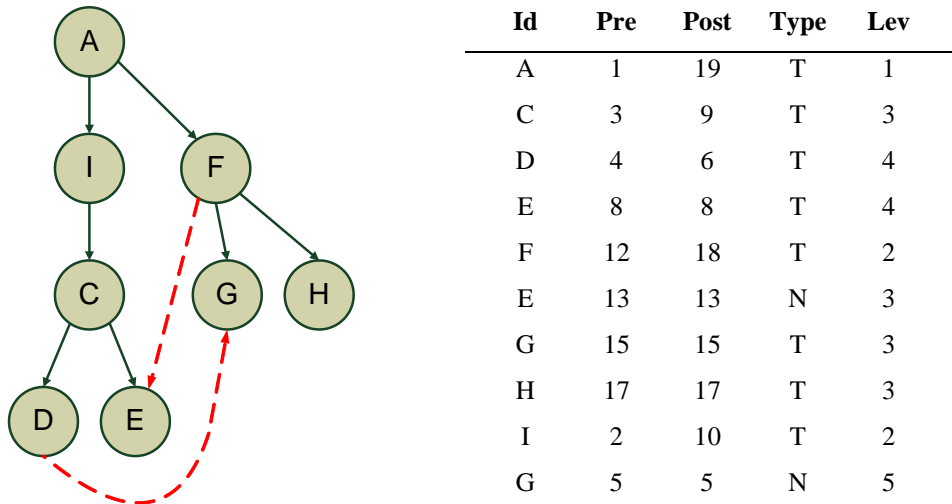
Fig. 7. General graph ontology and its tabular representation

(O5) Deletion of a non-tree edge. The operation **DeleteEdge(M,N)** can be used to remove the non-tree edge connecting node M to node N. Obviously, before deleting it, it must be verified that a non-tree edge from M to N actually exists and the tuple which encodes it (which has Id=N.Id, Pre<M.Pre, Post<M.Post, Type="N" and Lev=M.Lev+1) must be found. Its effects are, in practice, the reverse of the operation **(GO4)**. An algorithm for updating the tabular representation is the following:

```
DeleteEdge(M,N:GraphRow)
   DeleteRow(N.Id,N.Post,N.Post,N.Typ,N.Lev)
   ForEach Node in GraphTable Do
     If Node.Post>=M.Post
     Then Node.Post-=2
          If Node.Pre>M.Pre
          Then Node.Pre-=2 EndIf
     EndIf
   EndFor
Return
```

Starting from the ontology version in Fig. 7, we exemplify the effects of this operation with the execution of **DeleteEdge(F,E)**, which produces the final ontology version displayed in Fig. 8.

Id	Pre	Post	Type	Lev
A	1	17	T	1
C	3	9	T	3
D	4	6	T	4
E	8	8	T	4
F	12	16	T	2
G	13	13	T	3
H	15	15	T	3
I	2	10	T	2
G	5	5	N	5
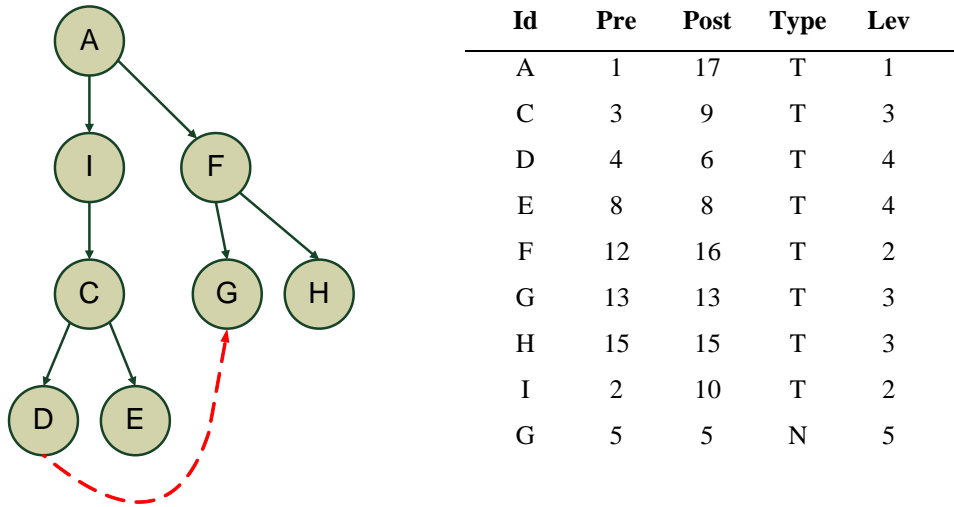
Fig. 8. General graph ontology and its tabular representation

3.2 Storing a Multi-version Ontology in a Temporal Relation

In this subsection, we show how the whole structural evolution of a general ontology can be represented and maintained as a temporal relation storing all the time-stamped ontology versions. As we did in [10], we assume valid time [13] is used as time dimension, which allows ontology designers to also apply retro- and pro-active modifications. However, the adoption of transaction time [13] (in a transaction-time or bitemporal relation) would require simple modifications to the proposed management. Hence, a multi-version ontology can be stored in a temporal relation with schema:

GraphRelation(Id, Pre, Post, Type, Lev, From, To)

where tuples like the ones considered in Sec. 3.1 are augmented with the timestamping attributes From and To, representing the boundaries of a right-open time interval [From,To), which embodies the validity of the tuple. Such relation can be stored in a relational database and manipulated via SQL statements [16]. Although we adopted valid time, when ontology modifications are applied, we allow users to only specify the validity start of the modification itself. The validity end is assumed by default to be UC, which can be interpreted as a currently unknown time greater than the validity start, so that the newly created ontology version is always the last one on the time line. In practice, this corresponds to use valid time almost as if it were transaction time, with the added value of a degree of freedom concerning the validity start of the new ontology version, which does not necessarily coincide with the system time when the modification is applied. Hence, we can work today on the modifications which produce a new ontology version which will be put in operation, for example, at the beginning of the next week (proactive modification)[8]. In case the modification is retroactive, that is valid from a time T in the past, the new ontology version will supersede the version valid at T and all the versions which possibly followed such version. Notice that retroactive modifications are sometimes required for instance in the legal field, where a new law may state the retroactive modification of other previous laws, but might be necessary also in other application domains including the medical field (e.g., when maintainers cannot help but delay the execution of updates after the date they should have become effective).

Before applying any other modification, an empty GraphRelation temporal table must be initialized via the creation of the ontology root node by means of a call to the following procedure:

```
CreateRoot(T:TimePoint)
   { INSERT INTO GraphRelation
     VALUES (NewId(),1,1,1,'T',T,'UC') }
```

[8] Although other authors [14] proposed to use "Now" as the endpoint of a right-unlimited valid time interval and reserve "UC" for transaction time, we prefer to use "UC" for valid time too, in order to correctly represent the effects of proactive modifications. It would not make much sense indeed to represent the validity of some fact as [T,Now) when T is a future validity start.

```
Return
```

The type "T" assigned to the first node makes it the root of the spanning tree structure underlying the ontology (not to be confused with T, which is the time value to be assigned as validity start to the first ontology version).

The algorithms presented in Sec. 3.1 for the maintenance of ontologies in their tabular representation translate into the procedures which are listed in the rest of this subsection, where embedded SQL statements are also used to manage the ontology stored in the GraphRelation temporal table. Whereas we need a database relation to store all the versions of an ontology, we assume all the tuples making up a single ontology version can reasonably be kept in a buffer pool and, thus, further manipulated in main memory. For example, we will make use of a kind of snapshot query [13]:

```
SELECT * INTO GraphCursor
FROM GraphRelation WHERE To='UC'
```

which extracts the current snapshot from the temporal relation GraphRelation to fetch all the tuples belonging to the ontology consolidated version, which can then be accessed through a cursor GraphCursor in main memory, for further processing by the modification procedures, via SQL statements, or one tuple at a time within a ForEach loop (for simplicity, we do not enter into the detail of cursor operations).

The five procedures corresponding to the ontology maintenance algorithms presented in Sec. 3.1 are listed in the following (using pseudo-code with embedded SQL statements). Procedures have a second argument, T, representing the validity start of the modification.

For the insertion of a new leaf node, the algorithm presented in Sec. 3.1 for operation **(O1)** becomes as follows:

```
InsertUnder(NId:Id,T:TimePoint)
  { SELECT * INTO GraphCursor FROM GraphRelation
     WHERE To='UC';
    SELECT * INTO N FROM GraphCursor
     WHERE Id=NId AND Type='T';
    INSERT INTO GraphRelation
```

```
       VALUES (NewId(),N.Post+1,N.Post+1,'T',
              N.Lev+1,T,'UC') }
   ForEach Node in GraphCursor Do
      New.Pre:=Node.Pre;
      New.Post:=Node.Post;
      If Node.Post>=N.Post
      Then Node.To:=T; New.Post+=2;
            If Node.Pre>N.Pre Then New.Pre+=2 EndIf
      EndIf
      If Node.To=T
      Then { UPDATE GraphRelation SET To=T
               WHERE Id=Node.Id AND Type='T'
                 AND From=Node.From;
             INSERT INTO GraphRelation
             VALUES (Node.Id,New.Pre,New.Post,'T',
                     Node.Lev,T,'UC') } EndIf
   EndFor
Return
```

Notice that, in the ForEach loop, testing of the condition Node.To=T in the last If statement of the loop body is made to check whether Node has been modified, in order to apply the modifications to the GraphRelation (the modified tuple representing the old node version is assigned T as validity end and a new tuple representing the node new version is appended). After the ForEach loop, the tuple representing the newly created node is also appended.

For the insertion of an intermediate node within the class hierarchy, the procedure introduced in Sec. 3.1 as **(O2)** becomes:

```
InsertOver(NId:Id,T:TimePoint)
  { SELECT * INTO GraphCursor FROM GraphRelation
     WHERE To='UC';
    SELECT * INTO N FROM GraphCursor
     WHERE Id=NId AND Type='T';
    INSERT INTO GraphRelation
    VALUES (NewId(),N.Pre,N.Post+1,'T',
```

```
                    N.Lev,T,'UC') }
   ForEach Node in GraphCursor Do
      New.Pre:=Node.Pre;
      New.Post:=Node.Post;
      New.Lev:=Node.Lev;
      If Node.Post>N.Post
      Then Node.To:=T; Node.Post+=2
        If Node.Pre>N.Pre Then Node.Pre++ EndIf
      Else If Node.Pre>=N.Pre
      Then Node.To:=T; Node.Pre++; Node.Post++; Node.Lev++
      EndIf
      If Node.To=T
      Then { UPDATE SET To=T
                WHERE Id=Node.Id AND Type='T'
                   AND From=Node.From;
                INSERT INTO GraphRelation
                VALUES (Node.Id,New.Pre,New.Post,'T'
                         New.Lev,T,'UC') } EndIf
   EndFor
}
Return
```

The procedure deriving from the algorithm in Sec. 3.1 to be used for the deletion of a node (**O3**) is as follows:

```
DeleteNode(NId:Id,T:TimePoint)
  { SELECT * INTO GraphCursor FROM GraphRelation
     WHERE To='UC';
    SELECT * INTO N FROM GraphCursor
     WHERE Id=NId AND Type='T';
    UPDATE GraphCursor SET To=T
     WHERE Id=N.Id AND To='UC' }
  ForEach Node in GraphCursor Do
     New.Pre:=Node.Pre;New.Post:=Node.Post;
     New.Type:=Node.Type;New.Lev:=Node.Lev;
     If Node.Pre>N.Pre
```

```
      Then If Node.Post<N.Post
            Then Node.To:=T;Node.Pre--;
                  Node.Post--;Node.Lev--
            Else If Node.Post>N.Post
                  Then Node.To:=T;Node.Pre-=2;Node.Post-=2
                  EndIf
      Else If Node.Post>N.Post
            Then Node.To:=T;Node.Post-=2 EndIf
      If Node.To=T
      Then { UPDATE GraphRelation SET To=T
              WHERE Id=Node.Id and From=Node.From;
            INSERT INTO GraphRelation
            VALUES (Node.Id,New.Pre,New.Post,New.Type,
                    Node.Lev,T,'UC') } EndIf
  EndFor
Return
```

The procedure to be used for the creation of a new non-tree edge according to the algorithm **(O4)** in Sec. 3.3 becomes as follows:

```
InsertEdge(MId,NId:Id,T:TimePoint)
  { SELECT * INTO GraphCursor FROM GraphRelation
     WHERE To='UC';
    SELECT * INTO M FROM GraphCursor
     WHERE MId=NId AND Type='T';
    SELECT * INTO N FROM GraphCursor
     WHERE Id=NId AND Type='T';
    INSERT INTO GraphRelation
    VALUES (N.Id,M.Post+1,M.Post+1,'N',
            M.Lev+1,T,'UC') }
  ForEach Node in GraphCursor Do
    New.Pre:=Node.Pre;New.Post:=Node.Post;
    If Node.Post>=M.Post
    Then Node.To:=T; New.Post+=2;
         If Node.Pre>M.Pre Then New.Pre+=2 EndIf
    EndIf
```

```
      If Node.To=T
      Then { UPDATE GraphRelation SET To=T
               WHERE Id=Node.Id AND Type='T'
                 AND From=Node.From;
             INSERT INTO GraphRelation
             VALUES (Node.Id,New.Pre,New.Post,'T',
                       Node.Lev,T,'UC') } EndIf
  EndFor
Return
```

Finally, the procedure deriving from the algorithm **(O5)** in Sec. 3.3 to be used for the deletion of a non-tree edge is as follows:

```
DeleteEdge(MId,NId:Id,T:TimePoint)
  { SELECT * INTO GraphCursor FROM GraphRelation
      WHERE To='UC';
    SELECT * INTO M FROM GraphCursor
     WHERE Id=MId AND Type='T';
    SELECT * INTO N FROM GraphCursor
     WHERE Id=NId AND Type='N' AND M.Pre<Pre
       AND Post<M.Post AND Lev=M.Lev+1;
    UPDATE GraphRelation SET To=T
     WHERE M.Pre<Pre AND Post<M.Post
       AND Lev=M.Lev+1 AND Type='N'}
  ForEach Node in GraphCursor Do
    New.Pre:=Node.Pre;New.Post:=Node.Post;
    New.Type:=Node.Type;New.Lev:=Node.Lev;
    If Node.Post>M.Post
    Then Node.To:=T;Node.Post-=2;
         If Node.Pre>M.Pre
         Then Node.To:=T;Node.Pre-=2 EndIf
    EndIf
    If Node.To=T
    Then { UPDATE GraphRelation SET To=T
             WHERE Id=Node.Id and From=Node.From;
           INSERT INTO GraphRelation
```

```
             VALUES (Node.Id,New.Pre,New.Post,New.Type,
                    Node.Lev,T,'UC') } EndIf
  EndFor
Return
```
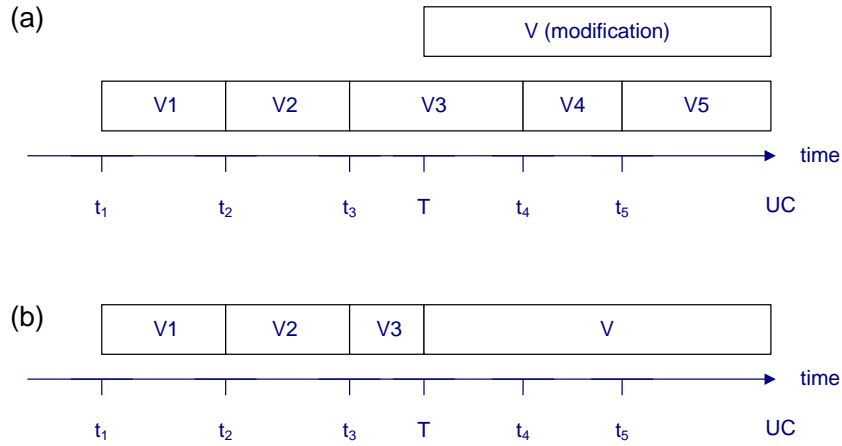
For the sake of simplicity, in writing the code, we assumed so far that only one ontology version is affected by the modification (i.e., the one with To equal to UC, which is part of the consolidated version valid at present time, further assuming that no versions with From>Now are currently stored in the GraphRelation temporal table). Otherwise, if more than one version can be affected, the SQL SELECT which loads GraphCursor at the beginning of the three procedures must be replaced by the SQL statements which follow:

```
DELETE FROM GraphRelation WHERE From>T;
SELECT * INTO GraphCursor FROM GraphRelation
 WHERE From<=T AND T<To
```

In fact, the creation of a new version valid from T involves all the tuples whose timestamp is totally or partially overlapped by the validity of the modification [T,UC). The validity start T is less than, equal to or greater than the current time Now in case of a retroactive, on time or proactive modification, respectively. In order to clarify what happens when a modification is applied to the history of an object in the most general case, we can consider the graphical example shown in Fig. 6.

Fig. 9. Effects of a modification in the most general case

In particular, Fig. 9(a) displays the history of an object composed of five versions (i.e., Vi valid from t_i to t_{i+1}, i=1..4, and V5 valid from t_5) and the placement on the time axis of a modification which must be applied to the history to produce a new version with contents V and validity from T. As shown in Fig. 9(b) presenting the history of the object after the application of the modification, the modification left versions V1 and V2 untouched, completely overlapped versions V4 and V5 (which have been removed) and partially overlapped version V3, whose validity has, thus, been restricted to $[t_3,T)$. After the deletion of the completely overlapped versions, the version affected by the modification is the one whose timestamp contains the validity start of the modification (i.e., V3 in Fig. 6(a), as $t_3 \leq T < t_4$). Therefore, the two SQL statements listed above accomplish the tasks, respectively, of deleting all the completely overlapped versions and of putting all the partially overlapped versions into GraphCursor for further processing by the procedures. Notice that the code of the procedures could indeed be easily changed in order to derive the new ontology version by applying the modification to the consolidated version (i.e., V5 in Fig. 9(a)) also when the modification is retroactive.

Finally, the proposed procedures can also easily be adapted in order to apply a sequence of primitives all valid from the same time T, so that a new ontology version

can be derived from the previous one by "simultaneously" applying more than one ontology modifications within a single transaction.

Coming back to the running example introduced in Sec. 3.1, we can easily store the first ontology version depicted in Fig. 1, assuming it has been created with validity starting at time T0, in the temporal relation displayed as Table 1.

Table 1. Temporal relation storing the first version of the ontology as appearing in Fig. 3

Id	Pre	Post	Type	Lev	From	To
A	1	15	T	1	T0	UC
B	2	2	T	2	T0	UC
C	4	8	T	2	T0	UC
D	5	5	T	3	T0	UC
E	7	7	T	3	T0	UC
F	10	14	T	2	T0	UC
E	11	11	N	3	T0	UC
G	13	13	T	3	T0	UC

Then we can consider the application of the following sequence of modifications which correspond to the ontology updates exemplified in Sec. 3.1:

InsertUnder(F,T1);
InsertOver(C,T2);
DeleteNode(B,T3);
InsertEdge(D,G,T4);
DeleteEdge(F,E,T5);

For instance, Table 2 shows the contents of the GraphRelation temporal table after the execution of **InsertUnder(F,T1)** on the initial state in Table 1. Notice how nodes A and F are represented through two tuples each, representing their versions belonging to the two ontology versions, respectively (e.g., the first version of A with preorder 1, postorder 15 and validity [T0,T1) belongs to the first ontology version, whereas the second version of A with postorder changed to 17 and validity [T1,UC) belongs to the second ontology version). Nodes represented through a single tuple (e.g., B) have a single version with validity [T0,UC) shared by both ontology versions. Obviously, the newly created node (H) has a single version with validity [T1,UC) belonging to the second ontology version only. We apologize with readers familiar with temporal databases, for whom such explanations might seem trivial.

Table 2. GraphRelation after the creation of leaf node H (it contains the first and second versions of the ontology in Fig. 3)

Id	Pre	Post	Type	Lev	From	To
A	1	15	T	1	T0	T1
B	2	2	T	2	T0	UC
C	4	8	T	2	T0	UC
D	5	5	T	3	T0	UC
E	7	7	T	3	T0	UC
F	10	14	T	2	T0	T1
E	11	11	N	3	T0	UC
G	13	13	T	3	T0	UC
A	1	17	T	1	T1	UC
F	10	16	T	2	T1	UC
H	15	15	T	3	T1	UC

We skip intermediate steps and show as Table 3 the contents of GraphRelation after the execution of the last **DeleteEdge(F,E,T5)** operation. The final state of the GraphRelation displayed as Table 3 contains all the six ontology versions, fully exemplifying the storage of our multi-version ontology in a single temporal relation.

Table 3. GraphRelation after the execution of all the five modifications (it contains all the six versions of the ontology in Fig. 3)

Id	Pre	Post	Type	Lev	From	To
A	1	15	T	1	T0	T1
B	2	2	T	2	T0	T3
C	4	8	T	2	T0	T2
D	5	5	T	3	T0	T2
E	7	7	T	3	T0	T2
F	10	14	T	2	T0	T1
E	11	11	N	3	T0	T2
G	13	13	T	3	T0	T2
A	1	17	T	1	T1	T2
F	10	16	T	2	T1	T2
H	15	15	T	3	T1	T2
A	1	19	T	1	T2	T3
C	5	9	T	3	T2	T3
D	6	6	T	4	T2	T3
E	8	8	T	4	T2	T3
F	12	18	T	2	T2	T3
E	13	13	N	3	T2	T3
G	15	15	T	3	T2	T3
H	17	17	T	3	T2	T3
I	4	10	T	2	T2	T3
A	1	17	T	1	T3	T4
C	3	7	T	3	T3	T4
D	4	4	T	4	T3	T4
E	6	6	T	4	T3	T4

F	10	16	T	2	T3	T4
E	11	11	N	3	T3	T4
G	13	13	T	3	T3	T4
H	15	15	T	3	T3	T4
I	2	8	T	2	T3	T4
A	1	19	T	1	T4	T5
C	3	9	T	3	T4	UC
D	4	6	T	4	T4	UC
E	8	8	T	4	T4	UC
F	12	18	T	2	T4	T5
E	13	13	N	3	T4	T5
G	15	15	T	3	T4	T5
H	17	17	T	3	T4	T5
I	2	10	T	2	T4	UC
G	5	5	N	5	T4	UC
A	1	17	T	1	T5	UC
F	12	16	T	2	T5	UC
G	13	13	T	3	T5	UC
H	15	15	T	3	T5	UC

Considering then the execution of a classical snapshot query [13] (retrieving the relation snapshot valid at time T and projecting out the timestamping attributes):

```
SELECT Id,Pre,Post,Lev FROM GraphRelation
WHERE From<=T AND T<To
```

over the temporal relation in Table 3, we can notice that:

— if T∈ [T0,T1), the retrieved snapshot coincides with the table in Fig. 1 (version 1);

— if T∈ [T1,T2), the retrieved snapshot coincides with the table in Fig. 3 (version 2);

— if T∈ [T2,T3), the retrieved snapshot coincides with the table in Fig. 4 (version 3);

— if T∈ [T3,T4), the retrieved snapshot coincides with the table in Fig. 5 (version 4);

— if T∈ [T4,T5), the retrieved snapshot coincides with the table in Fig. 6 (version 5);

— if T>T5, the retrieved snapshot coincides with the table in Fig. 7 (version 6).

This clearly highlights how the temporal relation in Table 3 is actually both a comprehensive and compact representation and a suitable storage scheme for the class hierarchy structure of a multi-version ontology.

As far as indexing of multi-version resources by means of references to ontology classes is concerned, we can underline the fact that the solution —used for the sake of simplicity in [1,5]— based on the bookkeeping of a single ontology version (i.e., the

consolidated version) to index all resource versions *is very inefficient from a practical point of view*, besides being simplistic and rather incorrect from a semantic and application requirement point of view [8]. As a matter of fact, a reference ontology might have to be used to index a very large repository of multi-version resources in a realistic environment. Hence, when even small changes (e.g., addition or deletion of a single class) are applied to an ontology in this scenario, where *preorder codes are used as class identifiers*, a large number of classes in the ontology might have their identifiers changed as a consequence of the update.Thus, such a change in class identifiers has to be propagated to all resources in order to preserve the correct semantic indexing, which would require to access and rewrite a large fraction of the whole resource repository (say, terabytes of data) to update class identifiers. With the *indirect reference* solution proposed in this work, ontology changes only affect the corresponding GraphRelation table (say, a few kilobyes of data) and do not require any changes to be applied to the resource repository.

4 PERSONALIZATION QUERY PROCESSING WITH MULTI-VERSION ONTOLOGIES

The semantic indexing of resources which link their contents to reference ontologies, as shown in Sec. 2, is designed to support personalization queries [1,5]. For this purpose, and in order to show how query processing works in the presence of a multi-version ontology, we consider the XQuery-like [17] query template which follows:

```
FOR $x IN resources.xml
WHERE TEXT_CONSTRAINT(XP,Key)
  AND VALID(T) AND APPLICABLE(Cx:depth)
RETURN $x
```

which is a slightly simplified form of the template introduced in [1,5] and for which a query engine has been implemented in a prototype system. The function `TEXT_CONSTRAINT()` applies textual constraints to the contents of the resources to be retrieved. Textual constraints can include both structural and lexical constraints, involving an XPath expression [18] which can be used for matching keywords within

the resource structured contents. For instance, the expression "TEXT_CONSTRAINT('//foo/bar/text()','baz AND qux')" selects the XML documents containing anywhere an element "foo" having a subelement "bar" whose contents include both the keyword "baz" and the keyword "qux". The function VALID() effects a temporal slicing of the resources by selecting the content versions valid at time "T". The function APPLICABLE() effects a sort of semantic slicing of the resources by selecting the content versions that are applicable to instances of ontology class "Cx" and of its ancestors up to "depth" levels. The meaning of the depth parameter has been also briefly recalled in Sec. 2. In [1,5], the expression "Cx:depth" has been called a *navigational pattern* with respect to the reference ontology.

In the presence of multi-version ontologies, the first step in query processing is the determination of the ontology classes denoted by the navigational pattern "Cx:depth" and of the preorder and postorder codes of such classes in the ontology version of interest. This information can be retrieved from the GraphRelation temporal table which stores the encoding of the multi-version ontology structure. For this purpose, first of all we assume to retrieve, for further processing, the ontology version valid at time "T" into a cursor OntologyCursor by means of the SQL snapshot query which follows:

```
SELECT Id,Pre,Post,Lev INTO OntologyCursor
FROM GraphRelation
WHERE From<=T AND T<To
```

Efficient execution of such preliminary operation can be supported by means of a suitable temporal access structure (like the RABTree index we proposed in [19]). Then, as in Sec. 3.2, we further assume that the whole contents of OntologyCursor can be kept in a buffer pool and, thus, further operations effected on its tuples, via formal SQL queries or cursor manipulation primitives, are fast operations working in main memory.

Hence, the next step of query processing takes two different ways according to the fact that non-tree edges of the underlying ontology are involved or not. In fact, if the

user-supplied navigation pattern does not involve nodes with incoming non-tree edges, query processing follows the same lines as described in [10] for tree-like ontologies. We will illustrate such a case first, in Sec. 4.1, since it is preliminary to the most general case also involving the traversal of non-tree edges which will be dealt with in Sec. 4.2.

4.1 Query Processing not Involving Non-tree Egdes

Assuming the navigational pattern "Cx:depth" denotes a set of nodes only connected by tree edges in the ontology graph, we can omit testing of the Type attribute in the SQL statements which can be used to search the OntologyCursor temporal table during personalization query processing. Hence, as done in [10] with tree-shaped ontologies, the retrieval of CX data can be effected by means of the the SQL query which follows:

```
SELECT * INTO CX FROM OntologyCursor
WHERE Id=Cx;
```

In fact, the query retrieves the data of the class CX whose identifier is "Cx" in the ontology version valid at time "T" which has been stored into OntologyCursor. Owing to the hypothesis of "Cx:depth" being free of non-tree edges, class CX has a single ancestor CY, that can be reached in "depth" steps starting from CX. Hence, making use of the level information associated to nodes, we can retrieve the CY data by means of the following SQL query from the ontology snapshot valid at time "T":

```
SELECT * INTO CY
FROM OntologyCursor AS Node
WHERE Node.Pre<CX.Pre AND Node.Post>CX.Post
  AND CX.Lev-Node.Lev=depth
```

The two ontology classes CX and CY retrieved in such a way must be used, in the second query processing step, to select the qualifying resource contents through their preorder and postorder codes. In particular, a resource version qualifies if its semantic pertinence implies the query navigational pattern [1,3,5]. Thanks to the properties of the preorder/postorder encoding, this notion of implication translates into verifying whether at least one of the ontology classes which make up the semantic pertinence of the resource is contained in a rectangular region defined in the preorder/postorder plane by the navigational pattern [1,5] (under the hypothesis of being free of non-tree edges, the "Cx:depth" navigational pattern individuates a single rectangular region in the preorder/postorder plane). Such rectangular region, in which all and only the nodes in the inheritance path from CY to CX fall, can be determined as the Cartesian product [CY.Pre,CX.Pre] x [CX.Post,CY.Post] (i.e., the lower right corner of the rectangle is CX, whereas the upper left corner is CY). Owing to the fact that preorder, postorder and level codes associated to the same classes are different in different ontology versions, we will have a different containment relationship to be checked for each ontology version. Notice that, in case the parameter "depth" is "0", the region denoted by the navigational pattern "Cx:0" reduces to a single point CX in the preorder/postorder plane and the containment tests reduce to equality tests.

For example, let us consider the multi-version ontology stored in our sample GraphRelation displayed in Table 3 and the query navigational pattern "D:2". Depending on the time "T" of interest, the CX and CY values retrieved by the above two queries navigating the ontology will be as summarized in Table 4.

Let us further consider the resource chunk in Fig. 2. The element foo(v1) is applicable to class B in [T1,T2); the element foo(v2) is applicable to class C in [T2,UC); the element foo/bar(v1) is applicable to class C or class G in [T2,UC). The relative positioning of such resource pertinences with respect to the regions individuated by

the navigational pattern "D:2" in the preorder/postorder plane for different time values is displayed in Fig. 10.

Table 4. Evaluation of the navigational pattern "D:2" in different temporal versions of the ontology stored as in Tab. 3

Time	CX			CY		
	Id	Pre	Post	Id	Pre	Post
[T0, T1)	D	5	5	A	1	15
[T1, T2)	D	5	5	A	1	17
[T2, T3)	D	6	6	I	4	10
[T3, T4)	D	4	4	I	2	8
[T4, T5)	D	4	6	I	2	10
[T5, UC)	D	4	6	I	2	10

Hence, at any time $T \in [T0,T1)$ there are no contents in the considered resource chunk which can qualify (the navigational pattern "D:2", from CX=D to CY=A upwards, translates into the [CY.Pre,CX.Pre] x [CX.Post,CY.Post] = [1,5] x [5,15] region, owing to the data in the first row of Tab. 4). At any time $T \in [T1,T2)$ the only valid element is foo(v1), which does not qualify since its semantic pertinence is class B (which has coordinates (2,2) in the preorder/postorder plane), which lays outside of the region individuated by the navigation pattern "D:2" (which is [1,5] x [5,17] in the plane, according to the second row of Tab. 4). At any time $T \in [T2,UC)$, valid elements are foo(v2) whose semantic pertinence is class C and its subelement foo/bar(v1) which inherits the applicability class C from its parent and also has applicability class G. Hence both foo(v2) and foo/bar(v1) qualify for the query in [T2,UC) as their pertinence class C is contained in the region individuated by the navigational pattern "D:2" (whereas the other applicability class G of foo/bar(v1) lays outside). In particular, the coordinates of C and the region individuated by "D:2" are, respectively, (5,9) and [4,6] x [6,10] in [T2,T3), (3,7) and [2,4] x [4,8] in [T3,T4), (3,9) and [2,4] x [6,10] in [T4,UC). Therefore, considering the chunk in Fig. 2 as the only available resource, a query with navigational pattern "D:2" returns an empty result if the tem-

poral selection condition involves a time T<T2 and retrieves the second version of the element foo (inclusive of the only version of the subelement foo/bar) if the temporal selection involves a time T≥T2.

4.2   Query Processing with General Graph Ontologies

In general, in the presence of general graph ontologies also containing non-tree edges, the evaluation of a navigational pattern "Cx:depth" is a bit more complicate. In fact, there can be multiple paths that can be followed moving from CX "depth" steps upwards in any ontology version, taking into account both tree and non-tree edges. Whereas, as exemplified in the previous subsection, the path composed of tree edges only gives rise to a single rectangular region in the preorder/postorder plane, other paths also containing non-tree edges may exist and give rise to multiple rectangular regions in the preorder/postorder plane. Let us still identify a preorder/postorder region be means of the pair (CX,CY) of its lower right and upper left corners, respectively. Hence we have, for instance, that the upward path C_1, C_2, C_3, C_4, C_5 gives rise, if C_3 is linked to C_2 by means of a non-tree edge (while the other ones in the path are tree edges), to two rectangular regions: (C_1,C_2) and (C_3,C_5). Thanks to the assumption that OntologyCursor can be manipulated in main memory and thanks to the fact that the "depth" parameter is a small number in any reasonable personalization query (e.g., we can assume with a very high confidence that "depth" usually ranges from 0 to 2), we can simply employ an iterative search method to retrieve all the desired regions. At each iteration, we navigate upwards one level of the ontology to locate direct ancestors (i.e., parents), either through tree or non-tree edges. Being (N_0,N) a region in the current set of working regions (RWork) and N' the direct ancestor of N in the ontology, if N is reached from N' via a tree edge, then the region can be extended as (N_0,N') to also include N' in the path; on the contrary, if it is reached via a non-tree edge, then a new region (N',N'), initially made of a single node, can be added to the current set of regions RWork. Obviously, if N' has been previously reached via another path, there is no point in continuing the extension of the current path from it (otherwise, we would redundantly have overlapping paths in the final result set). Hence,

the algorithm also takes into account, by means of the Visited node set, of the previously reached nodes in order to locally stop the path search. The current region is moved from RWork to the final result set Regions when it must not be extended further. At the end of the main loop, also the Regions left in RWork, which cannot be extended just because we run out of the "depth" steps, are added to the result set. Thus, the desired Regions set can be iteratively constructed as follows:

```
{ SELECT * INTO CX FROM OntologyCursor
    WHERE Id=Cx AND Type='T' }
Regions:={}; RWork:={(CX,CX)}; Visited:={CX}
For Step:=1 To depth Do
  ForEach Node In OntologyCursor Do
    ForEach (N0,N) In RWork Do
      If Node.Pre<N.Pre And N.Post<Node.Post
        And Node.Lev=N.Lev+1 And Node.Type='T' Then
          RWork-={(N0,N)};
          If Node In Visited
          Then Regions+={(N0,N)}
          Else RWork+={(N0,Node)}; Visited+={Node}
          EndIf
      EndIf
      If Node.Id=N.Id And Node.Type='N' Then
        { SELECT * INTO Node' FROM OntologyCursor
            WHERE Node'.Pre<Node.Pre AND
Node.Post<Node'.Post
              AND Node'.Lev=Node.Lev+1 AND
Node'.Type='T' }
        If Node' Not In Visited Then
          RWork+={(Node',Node')}; Visited+={Node'}
EndIf
      EndIf
    EndFor
  EndFor
EndFor
Regions+=RWork
```
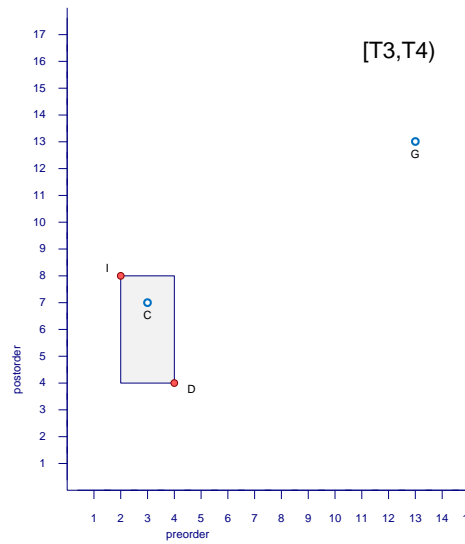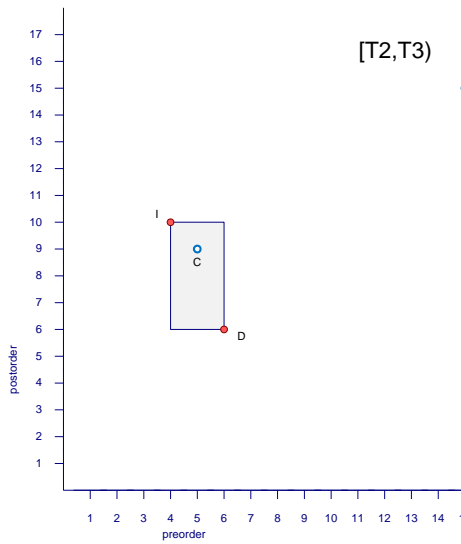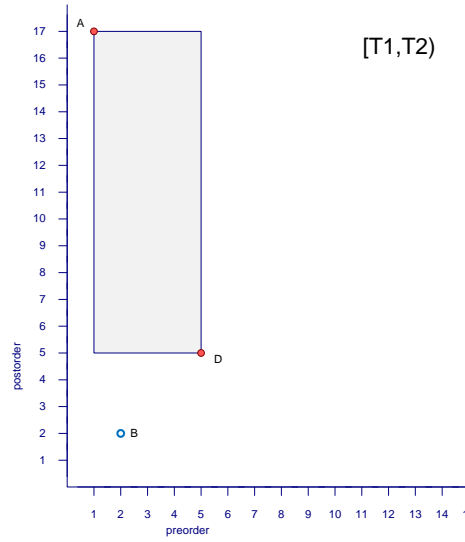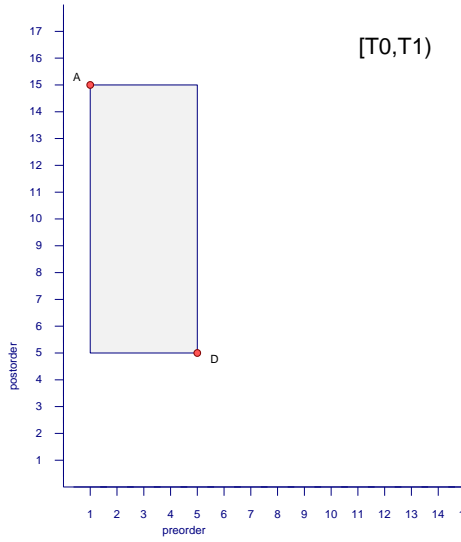
Notice that multiple tuples with the same Id may be present in OntologyCursor (remember, e.g., Fig. 8): one tuple with type "T", which contains preorder, postorder and level data of the node representing the ontology class identified by Id, and zero or more tuples with type "N", which actually represent the non-tree edges incoming in the ontology class identified by Id. This is the reason why the first SQL query looks for the only T-type tuple identified by Cx in order to initialize Regions, RWork and Visited variables. Then, at each step of the for loop on "depth", tuples involving predecessor nodes of N are initially determined as Node, owing to the ancestor-descendant relationship (1) and the condition that their levels differ by 1. If it can be reached from a tree-type edge (i.e., Node.Type="T"), then the tuple in Node represents a real ancestor node and can be processed as it is. If it has been reached from a non-tree-type edge (i.e., Node.Type="N", representing a *hop* node in the terminology of [15]), then the tuple representing the real node with the same Id of Node must be found (as Node' by the second SQL query) to be further processed.
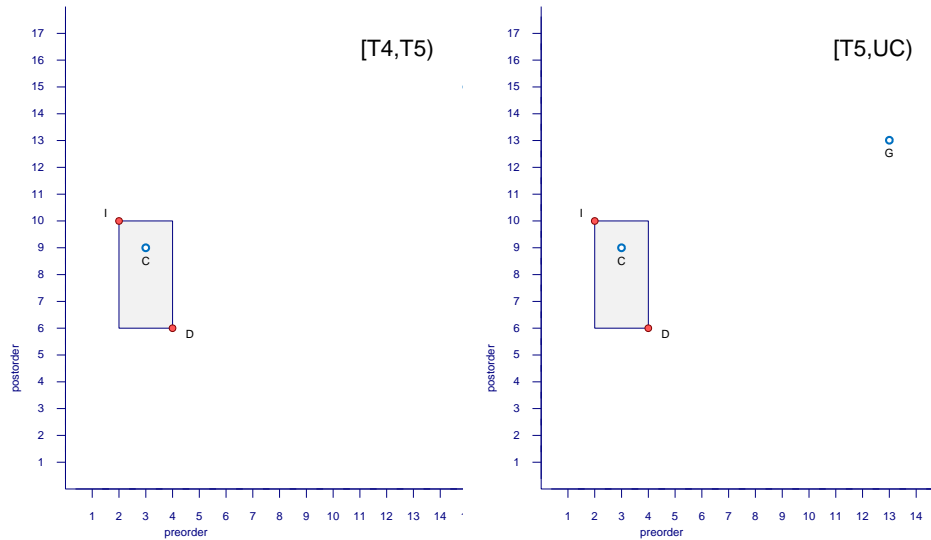
For instance, considering the ontology structure stored in Tab. 3, the navigational pattern "E:2" is evaluated by the above algorithm as denoting the regions {(E,A), (F,A)} in [T0,T2], {(E,I), (F,A)} in [T2,T5] and {(E,I)} in [T5,UC]. In fact, when T∈[T0,T1), the ontology version stored into OntologyCursor is the same as in Fig. 3; RWork is initializerd to {(E,E)} and Visited to {E}; in the first step of the algorithm, RWork and Visited become {(E,C)} and {C,E} due the execution of the first If (when Node=C) and then {(E,C),(F,F)} and {C,E,F} due to the execution of the second If (Node=C and Node'=F); in the second and last step Rwork and Visited become {(E,A),(F,F)} and {A,C,E,F} due to first If and then {(E,A),(F,A)} and {A,C,E,F} due to the second If (when Node'=A, already belonging to Visited). In practice, with reference to the graph in Fig. 3, the region (E,A) is constructed from E by navigating the hierarchy two steps upwards via tree edges only, the hop node F is reached from E in one step upwards via the non-tree edge and gives rise to the region (F,A) by navigating the hierarchy from F in another step upwards via a tree edge. In a similar way, the regions corresponding to "E:2" can be constructed also when T belongs to other time intervals.

After all the preorder/postorder rectangular regions involved by the navigational pattern "Cx:depth" have been determined by the above procedure and stored into
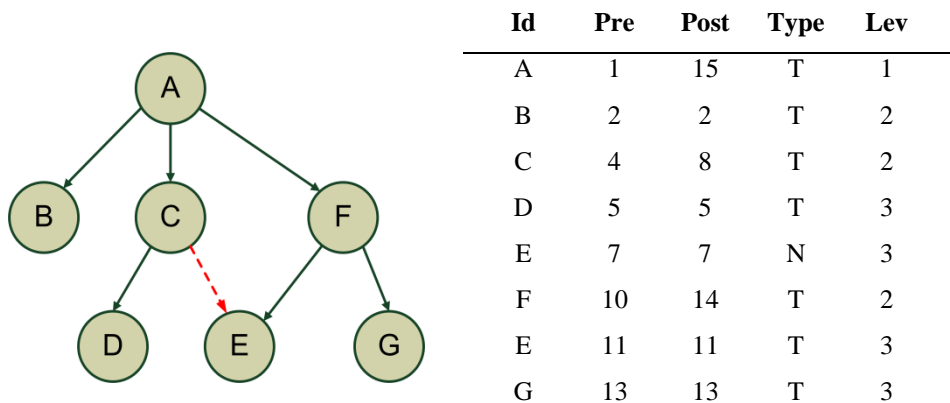
Regions in such a way, the second step of query processing, which requires to select the qualifying resource versions if their semantic pertinence implies the query navigational pattern, translates into verifying whether at least one of the ontology classes which make up the semantic pertinence of the resource is contained in at least one of the rectangular regions in Regions. Therefore, the only difference with respect to the previous case exemplified in Fig. 10, is that multiple rectangular regions to test inclusion may be present in this case, for any considered temporal version.

As far as the impact of non-tree edges on the overall query processing costs is concerned, it is unlikely to be sensibly relevant. First of all, the ontologies used in our personalization approach usually contain a limited number of non-tree edges. For instance, in the medical domain, the reference ontologies are mainly made of several independent taxonomies interconnected by some extra IS-A links [20] (the study [21] on SNOMED-CT evidenced that 27% only of the ontology classes have multiple parents). Ontologies used for personalization are semantically circumscribed subsets (e.g., diseases, hospital facilities, physician specialties) of general reference ontologies, including a few interconnected taxonomies isolated from the rest. Hence, due to the limited number of non-tree edges, to the fact that evaluation of a navigational pattern is performed in main memory and, in particular, to the usually short depth to be navigated, there is no need to resort to smart navigation algorithms and optimizations as those presented in [15] for general graphs.

[T0,T1)

[T1,T2)

[T2,T3)

[T3,T4)

Fig. 10. Query processing in the preorder/postorder plane. Placement of candidate resources is shown with blue circles

Id	Pre	Post	Type	Lev
A	1	15	T	1
B	2	2	T	2
C	4	8	T	2
D	5	5	T	3
E	7	7	N	3
F	10	14	T	2
E	11	11	T	3
G	13	13	T	3

Fig. 11**.** A different representation of the ontology structure in Fig. 3

Moreover, owing to the possible existence of multiple spanning trees in a graph, the same class hierarchy can be represented in different ways owing to different placements of the non-tree edges, which depend on the history of changes sequential-ly applied to build the ontology. For example, the same class hierarchy of Fig. 3 could be represented as shown in Fig. 11 indeed, with a non-tree edge connecting class C to

class E and a tree edge connecting class F to class E. Hence, one might question whether such diversity could have a noticeably impact on query costs. If a class C has n parents, one of them is connected by a tree edge and n-1 by non-tree edges. Thus, regardless of which incoming edge is the tree one, always one edge and n-1 non-tree edges have to be followed to navigate one level upwards. The actual placement of non-tree edges only influences the order in which nodes are visited and the way they are grouped in regions by the algorithm above (the same number of regions is anyway generated). For instance with the non-tree edge from C to E as in Fig. 11, regions {(E,A), (C,A)} would be generated for "E:2" in [T0,T2] instead of {(E,A), (F,A)}.

As outlined in [1; Sec. 3.6], in order to obtain a fully fledged efficient and scalable personalization engine, the selection of resource contents based on the semantic indexing described above can be combined with the holistic technology described in [1,22] and relying on the holistic temporal slicing techniques presented in [23]. In a few words, the holistic technology relies on a four-level architecture on which stack-based algorithms can be executed for efficient path and twig matching in querying an XML file [24]. For details on such an approach to personalization and for better characterization of usefulness of the personalization approach in the medical and legal domains, readers are referred to [1] and [22], respectively.

Finally, we can observe that the query template considered at the beginning of this section can easily be extended to support other temporal selection operators (e.g., to test overlap or containment of intervals) and to retrieve data valid over temporal intervals (i.e., also belonging to more than one temporal version of the resource) like in the more general formulation presented in [1]. Furthermore, also the applicability constraint can be extended to the general form presented in [1], where combinations with "AND" and "OR" logical operators of several navigational patterns in positive or negated form can be specified (in order to qualify for a negated navigational pattern, a resource must have its representative point outside the region defined by the navigational pattern in the preorder/postorder plane). Also applicability constraints involving multiple reference ontologies can be specified in the same query and processed as shown in [1].

5 RELATED WORKS

Information overload when searching a large repository for useful resources is a problem that can discourage, lead astray or at least hamper end users. Recommender systems [25,26], information filtering [27,28], personalized search [29,30] are examples of solutions, built upon the concept of user profile, which have been proposed to alleviate such problem, by increasing the quality and reducing the quantity of retrieved information and, thus, improving the search efficiency. In this context, *personalization* can be defined as the ways in which information and services can be tailored to match the unique and specific needs of an individual or a community [31]. Personalization is about understanding the needs of individuals and helping satisfy a goal that efficiently and knowledgeably addresses their needs in a given context [32]. Pretschner [33] presents a thorough discussion of ontology-based user profiling techniques and systems. More recent proposals, also in various application fields, include [34,35,36,37,38,39,40,41].

For details on the ontology-based personalization of (non-ontological) multi-version resources method we followed in this paper, or very similar approaches in the medical domain, we refer the reader to the already cited papers [1,2,3,4,5]. In fact, in such approaches, ontological information (e.g., concerning patient profiles or diagnostic facilities) is used to provide a selective access to a large repository of resources to produce a resource version tailored to a specific use case. Once the user interest has been focused on some ontology class(es), information on the class hierarchy can be used for selective access to annotated resources. When temporal resources are accessed, the necessity of also versioning the ontology has been identified (for the legal domain) in [8]. However, the technical problems which must be solved to efficiently support a full temporal perspective were not addressed before this work and its preliminary version [10].

The number of works dealing with ontology evolution and management of multiple ontology versions is quite large, as witnessed by the annotated bibliography introduced in [42]. Most relevant ones are referenced in the "Temporal Extensions of the Semantic Web" and "Versioning Issues" Sections of such a bibliography and carefully discussed in the "Ontology versioning approaches" Section of the excellent survey

[43]. Notice that, in this work, we do not deal with ontology evolution in a strict sense. We rather deal with evolution of an accessory data structure, embodying (or duplicating) the class IS-A hierarchy extracted from the ontology, which is necessary to support the personalized access to data resources. No other features of the underlying ontologies, including individuals and properties, are of interest for the (XML) data management problem considered here. This means that, in the presence of a fully fledged ontology, we assume its evolution can be managed, for instance, by means of one of the frameworks described in [43]; then the modifications involving the class hierarchy, which are the only of interest in our personalization approach, are collected (e.g., a list of them can be saved as a byproduct of the ontology evolution process, or they can even be extracted *a posteriori* by means of some change detection tool [42; Sec. 3.4, 43; Sec. 7.1.2]) to be applied to our accessory data structure, as it has been described in this work.

In order to efficiently manage the evolution of the ontology class hierarchy, we proposed to use a temporal relation. Several papers proposed efficient storage of ontologies in a relational database, including the SOR [44], Minerva [45] and ROSM [46] approaches. As far as those approaches are concerned: all of them can be adapted to store multi-version ontologies by making temporal all the relations in their proposed relational schema; they also provide relations for the storage and management of ontology instances, whereas the ontologies used for personalization as considered in this work (e.g., derived from the SNOMED-CT terminologies) are not equipped with instances; the only table, out of the fourteen in the schema of [46] and more than three dozens used in [44,45], which is relevant for the personalization method considered in this work is the one storing the "SubClassOf" information. The GraphRelation considered in this paper substitutes (or complements) the information stored in a "SubClassOf" relation made temporal. Hence, our approach in Section 3 can be thought as aimed at mapping high-level operations acting on the ontology class hierarchy onto low-level operations on time-stamped tuples encoding the ontology structure as stored in the GraphRelation.

On the other hand, several works deal with the representation and storage of temporal ontologies and propose, for instance, temporal extensions of RDF or OWL ontology languages [47,48,49,50,51]. Whereas the target of such approaches is the man-

agement of temporal versions of an evolving complete ontology (i.e., also including properties and instances), we once again emphasize that this work is complementary to such approaches, as it is focused on an accessory temporal data structure representing the evolution of the class hierarchy only, extracted from the ontology. Such data structure accompanies the main ontology, evolving in parallel and for which any of the proposed temporal RDF/OWL representation formats can be used, and is managed separately in order to efficiently support our proposed personalization method by means of the relational storage and management algorithms presented in Sec. 3 and 4.

6   CONCLUSIONS

In this paper, which extends the approach of [10] to general graph ontologies, we introduced a storage scheme which can be used to represent and manage in a temporal database the evolution of the class hierarchy of an ontology used for personalization. A multi-version ontology structure stored in a temporal relational table according to the proposed representation scheme can be manipulated via standard SQL queries. The definition of primitive operations, which can be used for the maintenance of the ontology structure in such a framework, has also been provided. Moreover, it has been shown how the query processing method described in [1,22] has to be augmented in order to deal with multi-version ontologies in the presence of the storage scheme presented in this work. Notice that previous approaches on ontology-based personalization did not take into account the availability of multiple ontology versions, whereas their exploitation to reconstruct a consistent past perspective on personalization is a necessity of advanced applications (e.g., in the legal and medical domains). We also released for the first time in this work the simplifying assumption of dealing with tree-shaped ontologies, on which our previous approaches to personalization [1,5], including [10], were built. In fact, we have shown, in Sec. 3, how the evolution of general graph ontologies can be managed and, in Sec. 4, how their adoption for personalization can also be supported.

In future work, we will also consider more thoroughly performance aspects of the proposed solutions. In particular, we will experimentally test the efficiency of the approach in the presence of very large ontologies, with thousands of classes and doz-

ens of versions each. In such a case, the adoption of a temporal index structure like the RABTree [19], which is a lean secondary index I/O-optimal in the absence of data duplication, might reveal crucial in order to cope with the size growth of the GraphRelation temporal table in the long run, with versions continuing to pile up at a stable rate. The impact of a multi-version ontology on personalization query processing costs will be also extensively studied on a future release of the prototype described in [1,5]. Preliminary results [52] (with tree-like ontologies) have shown that the solutions presented in this paper allow, with a minimal impact on the efficiency of the personalization engine, a very efficient management of ontology updates and consequent adaptation of indexed resources.

7   REFERENCES

[1]   F. Grandi, F. Mandreoli, R. Martoglia, Efficient management of multi-version clinical guidelines, J. Biomed. Inform. 45 (6) (2012) 1120–1136.

[2]   D. Riaño, F. Real, J.A. López-Vallverdú, F. Campana, S. Ercolani, P. Mecocci, R. Annicchiarico, C. Caltagirone, An ontology-based personalization of health-care knowledge to support clinical decisions for chronically ill patients, J. Biomed. Inform. 45 (3) (2012) 429–446.

[3]   S.W. Tu, M. Peleg, S. Carini, M. Bobak, J. Ross, D. Rubin, I. Sim, A practical method for transforming free-text eligibility criteria into computable criteria, J. Biomed. Inform. 44 (2) (2011) 239–250.

[4]   S. Zheng, F. Wang, J. Lu, Enabling Ontology Based Semantic Queries in Biomedical Database Systems, Int. J. Semantic Comp. 8 (1) (2014) 67–83.

[5]   F. Grandi, F. Mandreoli, R. Martoglia, E. Ronchetti, M.R. Scalas, P. Tiberio, Ontology-Based Personalization of E-Government Services, in: P. Germanakos, C. Mourlas (Eds.), Intelligent User Interfaces: Adaptation and Personalization Systems and Technologies, Information Science Reference Series, IGI Global, Hershey, 2009, pp. 167–187.

[6]   W3C Consortium, Extensible Markup Language (XML) Home Page, 2014. Retrieved September 1, 2014 from http://www.w3.org/XML/.

[7]   M.J. Field, K.N. Lohr (Eds.), Clinical Practice Guidelines: Directions for a New Program, National Academy Press, Washington, 1990.

[8]   F. Grandi, M.R. Scalas, The Valid Ontology: a Simple OWL Temporal Versioning Framework, in: 3rd Int. Conf. on Advances in Semantic Processing (SEMAPRO), IEEE Computer Society Press, Sliema, Malta, 2009, pp. 98–102.

[9]   T.K. Mackey, B.A. Liang, The Role of Practice Guidelines in Medical Malpractice Litigation. Virtual Mentor 13 (1) (2011) 36–41.

[10] F. Grandi, Dynamic Multi-version Ontology-based Personalization, in: 2nd Int. Workshop on Querying Graph Structured Data (GraphQ), ACM Press, Genoa, Italy, 2013, pp 224−232.

[11] P.F. Dietz, D.D. Sleator, Two Algorithms for Maintaining Order in a List, in: 19th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, Montreal, Québec, 1987, pp. 365–372.

[12] T. Grust, M. van Keulen, J. Teubner, Accelerating XPath evaluation in any RDBMS, ACM Trans. Database Syst. 29 (1) (2004) 91–131.

[13] C.S. Jensen, C.E. Dyreson, M.H. Böhlen, J. Clifford, R. Elmasri, S.K. Gadia, F. Grandi, P.J. Hayes, S. Jajodia, W. Käfer, N. Kline, N.A. Lorentzos, Y.G. Mitsopoulos, A. Montanari, D.A. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N.L. Sarda, M.R. Scalas, A. Segev, R.T. Snodgrass, M.D. Soo, A.U. Tansel, P. Tiberio, G. Wiederhold, The Consensus Glossary of Temporal Database Concepts - February 1998 Version, in: O. Etzion, S. Jajodia, S. Sripada (Eds.), Temporal Databases, Research and Practice, LNCS 1399, Springer Verlag, Berlin, 1998, pp. 367–405

[14] J. Clifford, C. Dyreson, T. Isakowitz, C.S. Jensen, R.T. Snodgrass, On the semantics of "now" in databases, ACM Trans. Database Syst. 22 (2) (1997) 171–214.

[15] S. Trißl, U. Leser, Fast and practical indexing and querying of very large graphs, in Proceedings of the 2007 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD), ACM Press, Beijing, China, 2007, pp. 845–856.

[16] R.T. Snodgrass, Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann, San Francisco, 1999.

[17] S. Boag D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation, 2011. Retrieved September 1, 2014 from http://www.w3.org/TR/xquery/

[18] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernández, M. Kay, J. Robie, J. Siméon, XML Path Language (XPath) 2.0 (Second Edition), W3C Recommendation, 2011. Retrieved September 1, 2014 from http://www.w3.org/TR/xpath20/.

[19] F. Grandi, Lean Index Structures for Snapshot Access in Transaction-time Databases, in: 21st Int. Symposium on Temporal Representation and Reasoning (TIME), IEEE Computer Society Press, Verona, Italy, 2014, pp. 91–100.

[20] L. Zhang, Y. Perl, M. Halper, J. Geller, J.J. Cimino, An Enriched Unified Medical Language System Semantic Network with a Multiple Subsumption Hierarchy, J. Am. Med. Inform. Assoc. 11 (3) (2004) 195–206.

[21] O. Bodenreider, B. Smith, A. Kumar, A. Burgun, Investigating subsumption in SNOMED CT: An exploration into large description logic-based biomedical terminologies, Artif. Intell. Med. 39 (3) (2007) 183–195.

[22] F. Grandi, F. Mandreoli, R. Martoglia, Issues in Personalized Access to Multi-version XML Documents, in: E. Pardede (Ed.), Open and Novel Issues in XML Database Applications, Information Science Reference Series, IGI Global, Hershey, 2009, pp. 199–230.

[23] F. Mandreoli, R. Martoglia, E. Ronchetti, Supporting Temporal Slicing in XML Databases, in: 10th Int. Conf. on Extending Database Technology (EDBT), LNCS 3896, Springer Verlag, Munich, Germany 2006, pp. 295–312.

[24] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: 2002 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD), ACM Press, Madison, WI, 2002, pp. 310–321.

[25] P. Resnick, H. R. Varian (Eds.), Special issue on Recommender systems, Commun. ACM 40 (3) (1997).

[26] S. Perugini, M. A. Gonçalves, E.A. Fox, Recommender Systems Research: A Connection-Centric Survey, J. Intell. Inf. Syst. 23 (2) (2004) 107–143.

[27] U. Hanani, B. Shapira, P. Shoval, Information Filtering: Overview of Issues, Research and Systems, User Model. User-Adapt. Interact. 11 (3) (2001) 203–259.

[28] D. Godoy, A. Amandi, User Profiling for Web Page Filtering, IEEE Internet Comput. 9 (4) (2005) 56–64.

[29] J. Pitkow, H. Schütze, T. Cass, R. Cooley, D. Turnbull, A. Edmonds, E. Adar, T. Breuel, Personalized search, Commun. ACM 45 (9) (2002) 50–55.

[30] A. Micarelli, F. Gasparetti, F., Sciarrone, S. Gauch, Personalized Search on the World Wide Web, in: P. Brusilovsky, A. Kobsa, W. Nejd (Eds.), The Adaptive Web: Methods and Strategies of Web Personalization, LNCS 4321, Springer Verlag, Berlin, 2007, pp. 195–230.

[31] J. Callan, A. Smeaton, M. Beaulieu, P. Borlund, P. Brusilovsky, M. Chalmers, C. Lynch, J. Riedl, B. Smyth, U. Straccia, E. Toms, Personalization and recommender systems in digital libraries, in: Joint NSF-EU DELOS Working Group Report, ERCIM, Sophia An-

tipolis, France, 2003. Retrieved September 1, 2014 from http://www.ercim.org/publication/ws-proceedings/Delos-NSF/Personalisation.pdf.

[32] D. Riecken, Personalized views of personalization. Commun. ACM 43 (8) (2000) 27–28.

[33] A. Pretschner, Ontology based personalized search. Unpublished master's thesis, University of Kansas, Lawrence, Kan, 1998.

[34] S. Gauch, J. Chaffee, A. Pretschner, Ontology-Based User Profiles for Search and Browsing, Web Intell. and Agent Syst. 1 (3-4) (2003) 219–234.

[35] L. Razmerita, A. Angehrn, A. Maedche, Ontology-based User Modeling for Knowledge management Systems, in: 9th Int. Conf. on User Modeling (UM), LNAI 2702, Springer Verlag, Johnstown, PA, 2003, pp. 213–217.

[36] S. E. Middleton, N. Shadbolt, D.C. De Roure, Ontological user profiling in recommender systems, ACM Trans. Inform. Syst. 22 (1) (2004) 54–88.

[37] J. Trajkova, S. Gauch, Improving ontology-based user profiles, in: 7th Int. Conf. on Computer-Assisted Information Retrieval (RIAO), C.I.D., Paris, France, 2004, pp. 380–389.

[38] A. Sieg, B. Mobasher, R.D. Burke, Learning Ontology-Based User Profiles: A Semantic Approach to Personalized Web Search, IEEE Intell. Inform. Bull. 8 (1) (2007) 7–18.

[39] M. Golemati, A. Katifori, C. Vassilakis, G. Lepouras, C. Halatsis, Creating an Ontology for the User Profile: Method and Applications, in: 1st Int. Conf. on Research Challenges in Information Science (RCIS), IEEE Computer Society Press, Ouarzazate, Morocco, 2007, pp. 407–412.

[40] I. Cantador, A. Bellogín, P. Castells, A Multilayer Ontology-based Hybrid Recommendation Model, AI Commun. 21 (2) (2008) 203–210.

[41] A. Moreno, A. Valls, D. Isern, L. Marina, J. Borràs, SigTur/E-Destination: Ontology-based personalized recommendation of Tourism and Leisure Activities, Eng. Appl. Artif. Intell. 26 (1) (2013) 633–651.

[42] F. Grandi, Introducing an Annotated Bibliography on Temporal and Evolution Aspects in the Semantic Web, ACM SIGMOD Rec. 41 (4) (2012) 18–21.

[43] F. Zablith, G. Antoniou, M. d'Aquin, G. Flouris, H. Kondylakis, E. Motta, D. Plexousakis, M. Sabou, Ontology Evolution: A Process Centric Survey, Knowl. Eng. Rev. 30 (1) (2015) 45–75..

[44] J. Lu, L. Ma, L., Zhang, J.-S. Brunner, C. Wang, Y. Pan, Y. Yu, SOR: A Practical System for Ontology Storage, Reasoning and Search, in: 33rd Int. Conf. on Very Large Data Bases (VLDB), ACM Press, Vienna, Austria, 2007, pp. 1402–1405.

[45] S. Heymans, L. Ma, D. Anicic, Z. Ma, N. Steinmetz, Y. Pan, J. Mei, A. Fokoue, A. Kalyanpur, A. Kershenbaum, E. Schonberg, K. Srinivas, C. Feier, G. Hench, B. Wetzstein, U.

Keller, Ontology Reasoning with Large Data Repositories, in: M. Hepp, P. De Leenheer, A. de Moor, Y. Sure (Eds.), Ontology Management, CHE 7, Springer Verlag, Berlin, 2008: pp. 89–128.

[46] Z. Zhou, X. Yongkang, A study on ontology storage based on relational database, in: 13Th IEEE Joint Int. Computer Science and Information Technology Conf. (JICSIT), IEEE Computer Society Press, Chongqing, China, 2011, pp. 1–5.

[47] C. Gutierrez, C. Hurtado, A. Vaisman, Introducing time into RDF, IEEE Trans. Knowl. and Data Engin. 19 (2) (2007) 207–218.

[48] A. Pugliese, O. Udrea, V.S. Subrahmanian, Scaling RDF with Time, in: 17th Int. World Wide Web Conf. (WWW), ACM Press, Beijing, China, 2008, pp. 605–614.

[49] J. Tappolet, A. Bernstein, Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL, in 6th Europ. Semantic Web Conf. (ESWC), LNCS 5554, Springer-Verlag, Heraklion, Greece, 2009, pp. 308–322.

[50] F. Grandi, Multi-temporal RDF Ontology Versioning, in: 3rd Int. Workshop on Ontology Dynamics (IWOD), CEUR-WS, Washington, DC, 2009. Retrieved September 1, 2014 from http://ceur-ws.org/Vol-519/grandi.pdf.

[51] V. Milea, F. Frasincar, U. Kaymak, tOWL: A temporal web ontology language, IEEE Trans. Syst. Man and Cybern., Part B 42 (1) (2012) 268–281.

[52] F. Grandi, F. Mandreoli, R. Martoglia, Multi-version ontology-based personalization of clinical guidelines, (*in preparation*).

Fabio Grandi received from the University of Bologna, Italy, a Laurea degree cum Laude in Electronics Engineering in 1988 and a Ph.D. in Electronics Engineering and Computer Science in 1994. From 1989 to 2002 he has worked at the CSITE center of the Italian National Research Council (CNR) in Bologna in the field of neural networks and temporal databases, initially supported by a CNR fellowship. In 1993 and 1994 he was an Adjunct Professor at the Universities of Ferrara, Italy, and Bologna. In the University of Bologna, he was with the Dept. of Electronics, Computers and Systems from 1994 to 1998 as Research Associate and as Associate Professor from 1998 to 2012, when he joined the Dept. of Computer Science and Engineering. He is currently an Associate Professor of Information Systems in the School of Engineering and Architecture of the University of Bologna. His scientific interests include temporal, evolution and versioning aspects in data management, WWW and Semantic Web, knowledge representation, storage structures and access cost models.