Efficient Management of Multi-version Clinical Guidelines

Fabio Grandi\textsuperscript{a}, Federica Mandreoli\textsuperscript{b}, Riccardo Martoglia\textsuperscript{c}

\textsuperscript{a} DEIS, University of Bologna, Viale Risorgimento 2, Bologna, Italy (e-mail: fabio.grandi@unibo.it).
\textsuperscript{b} DII, University of Modena and Reggio Emilia, Via Vignolese 905, Modena, Italy (e-mail: federica.mandreoli@unimo.it).
\textsuperscript{c} DII, University of Modena and Reggio Emilia, Via Vignolese 905, Modena, Italy (e-mail: riccardo.martoglia@unimo.it).
Abstract—Clinical medicine and health-care developments in recent years testified a tremendous increase in the number of available guidelines, i.e., “best practices” encoding and standardizing care procedures for a given disease. Clinical guidelines are subject to continuous development and revision by committees of expert physicians and health authorities and, thus, multiple versions coexist as a consequence of the clinical and healthcare activities. Moreover, several alternatives are usually included in order to make the guidelines as general as possible, making them difficult to handle both in manual and automated fashions. In this work, we will introduce techniques to model and to provide efficient personalized access to very large collections of multi-version clinical guidelines, which can be stored both in textual and in executable format in an XML repository. In this way, multiple temporal perspectives, patient profile and context information can be used by an automated personalization service to efficiently build on demand a guideline version tailored to a specific use case.

Index Terms—multi-version clinical guidelines, personalized access, temporal indexing, semantic indexing, XML repositories.

A preliminary version of this work has been presented at the Third International Conference on Health Informatics, Valencia, Spain, January 2010.
1. Introduction

Clinical guidelines, according to the definition endorsed by the American Institute of Medicine, are “systematically developed statements to assist practitioners and patient decisions about appropriate health care for specific clinical circumstances” [1]. In other words, clinical guidelines are best practices encoding and standardizing care procedures for a given disease [2]. According to [3], clinical medicine and health-care developments in recent years testified a tremendous increase in the number of available guidelines, which has occurred because health-care decision makers view guidelines as a method, supported by a growing body of evidence, to improve health outcomes, to reduce unjustified practice variation and, potentially, to contain costs. During the past 30 years, there have been several efforts to support guidelines-based care in an automated fashion. The advantages of adopting computer-based guidelines as a means for improving the work of physicians and optimizing hospital activities have been acknowledged by many authors and several systems have been developed (e.g., [4,5,6]), with the World Wide Web becoming an outstanding venue for user-friendly and improved access to guideline repositories (e.g., [7,8]). In addition to narrative guidelines, which can be searched and consulted through a retrieval system, a growing importance is gained by computer-interpretable guidelines, which can be executed to give decision support to care providers [9]. An executable guideline specification is also said to represent the conceptual model of its narrative counterpart.

Clinical guidelines are subject to continuous development and revision by committees of expert physicians and health authorities and, thus, multiple versions coexist as a consequence of the clinical and healthcare activities. Moreover, several alternatives are usually included in order to make the guidelines as general as possible, which sometimes makes them cumbersome and difficult to handle. How to choose among existing guidelines and how to effectively deliver the right versions to clinicians have become questions to be addressed. Relevant obstacles to their deployment and dissemination have been found in the need of tailoring guidelines to specific patient populations [3] and of adapting them to constraints in local settings, like available hospital resources and practitioners’ skills [10]. Last but not least, clinical guidelines have also been recently proposed to be used as evidence of the legal standard of care in medical malpractice litigation [11] and, thus, the knowledge of the right guideline version to be applied to a given patient case at a given time in a given context becomes more and more important.

In this work, we will introduce techniques to model and to provide efficient personalized access to very large collections of multi-version clinical guidelines, which can be stored both in textual and in executable format in an XML repository. The XML language [12] has been proposed by many authors and adopted in several research projects (e.g., [13,14,15,16]) as a suitable means to encode and spread over the Internet
clinical guidelines. In the DeGeL framework [17,6;pp. 203–212], a clinical guideline library equipped with a set of computer-aided support tools, the XML language is adopted as a medium to enable the specification of a guideline in different formalisms, such as ASBRU [18], GLIF [19], and GEM [14]. In particular, in the DeGeL hybrid modelling approach, multiple representations of a clinical guideline are provided at different representation levels: semi-structured, semi-formal, and formal; at each level, guidelines are encoded as XML documents (in ASBRU, GLIF or GEM format). Hence, our approach can be considered as a compatible extension of such proposals, to which we aim at adding multi-version representation capabilities and efficient personalization query facilities. In this way, multiple temporal perspectives, patient profile and context information can be used by an automated personalization service to build a guideline version tailored to a specific use case. Notice that some research efforts (e.g., [20,21,22]) also focus on the demanding tasks of automatic maintenance of state-of-the-art guidelines from published guideline variants, on one hand, or of automatic adaptation of existing guidelines to a new context to reduce costs and risks of a development from scratch, on the other hand. The problem we indeed want to solve is, once a new guideline version is created, whichever is its origin, how to integrate it with the old versions in a compact multi-version representation and store them together in a single repository, and how to efficiently extract any specific desired version from the repository on user demand. Notice that maintaining different versions as separate documents would be unfeasible, as their number along with storage space requirements will be subject to a combinatorial explosion due to the multidimensional nature of versioning.

A general framework in which our proposal could be embedded is, for instance, the DeGeL environment [17]. We choose it as reference framework mainly for two reasons: the availability in a comprehensive architecture of a quite complete suite of tools covering all the phases of the clinical guideline lifecycle, and a broad consensus gained from the medical community in several assessment trials. The weak point we individuated in such a framework, when we considered addition of guideline multi-versioning and personalization facilities, is the adopted storage back-end, which is supposed to be a relational DBMS (as it is also in [23], where adaptation issues are indeed considered). In our previous research [24], we showed how such an approach becomes definitely inadequate when the number of documents and versions per document increases since, in order to select the relevant parts and consistently assemble the desired versions, one document at a time must be processed and a large amount of main memory is required, which strongly limits scalability and concurrency. Therefore, in this work we focus on that weak point and, in order to overcome the highlighted limitations and make the management and personalization of multi-version guidelines feasible, we developed a novel modelling approach and query
processing technology. Consequently, we will evaluate our approach considering performance aspects only, by analyzing its efficiency and scalability properties. Nevertheless, we acknowledge that the assessment in a real setting of the clinical relevance of our versioning and personalization methods is also a fundamental milestone and, thus, will be the objective of future works.

The paper, after further related work is referenced in Subsection 1.1, is organized as follows. Section 2 analyzes the modelling of multi-version clinical guidelines: temporal and semantic versioning of clinical guidelines is introduced in Section 2.1 and 2.2, respectively, with reference to advanced application requirements; Section 2.3 is devoted to the description of a multidimensional XML data model supporting temporal and semantic versioning of guidelines; whereas Section 2.4 sketches how a guideline editing tool has to be extended in order to support modifications of a multi-version XML document, Section 2.5 describes the query interface adopted to extract guideline versions. In Section 3, we describe in details the data management infrastructure satisfying the complex requirements of an XML multi-version clinical guidelines personalization engine: we begin by providing some background in Section 3.1 and 3.2, then we discuss the foundations of the native XML personalization engine in Section 3.3; we introduce the adopted temporal and semantic indexing scheme in Section 3.4 and describe the flexible holistic technology we devised for personalized access to multi-version guideline repositories in Sections 3.5 and 3.6. Conclusions and future work directions will finally be found in Section 4.

1.1. Related Work

(Non-temporal) versioning and configuration issues have been previously considered in the field of design databases [25] and software engineering [26], whereas schema versioning [27] and its interaction with temporal data management have been studied in the context of traditional [28,29] and active databases [30]. Notice that versioning in such fields is usually motivated by the need of supporting non-destructive changes and concurrent updates, whereas personalized access using semantic information is not considered a primary issue. A consistent bibliography [31] is also devoted to temporal and versioning issues of XML data.

More strictly related work includes the management of norm documents in the legal domain, which presents similarities with clinical guidelines, for which we previously developed some techniques to deal with multidimensional versioning and efficiently support personalized access [32,33]. In the literature concerning clinical guidelines, some data models and computer tools have been proposed to manage versioning issues and to support adaptable clinical guidelines. In [34], Owens and Nease proposed a model to represent guidelines adaptable to variations in circumstances (patient populations or execution settings).
Guidelines are encoded as Markov decision models, which can be modified via the adjustment of numerical parameters to evaluate the cost-effectiveness of the adaptation. This approach served as building brick for the implementation of the ALCHEMIST system [3, 35], which allows developers to publish on the Web both a guideline and an interactive decision model. A computer tool, based on a Web interface and CGI scripts, enables them to tailor guidelines to a specific patient or patient population by trimming the values of input variables (or to highlight the potentially affected variables if a specific guideline element, like the patient population, is changed). In [36] the authors propose a practical method for transforming free-text eligibility criteria specified in clinical trials into computable criteria. The target is the ERGO annotation language and the method essentially consists of a semi-automated process based on natural language processing techniques followed by various algorithms that are used to transform ERGO annotations into computable expressions suitable for different use cases. The use case that relates to our work is “searching for studies enrolling specific patient populations” where ERGO annotations are formulates as OWL expressions. The CAMINO editing environment [37], based on the Protégé knowledge acquisition tool, is aimed at helping developers to graphically derive site-specific guidelines from generic ones. The HieroGLIF approach [38, 39] applies the Axiomatic Design principles to enable developers to encode in a hierarchical modular way setting-independent guidelines, which can then be adapted to local contexts. The approach is based on an authoring tool which extends the Java libraries developed for the GLIF project [19]. GLIF has also been extended for guideline versioning in [40], where methods based on log files, difference tables and version annotations have been considered. In particular, the introduction of version annotations allow the coexistence of several guideline versions in a single knowledge base while making the differences explicit. The capabilities of the GLIF3 Protégé-based authoring tool have been leveraged to enable developers to represent and visualize the changes applied to produce a version. The GLARE system [23, 41] provides a facility for automatic resource-based adaptation of guidelines to a specific execution context via pruning of non-applicable alternatives. It relies on the query capabilities of a standard DBMS for the management of guideline collections, while adaptation is effected through main-memory processing of the retrieved guidelines. In the ASBRU approach [18], preferences can be specified in order to bias or constrain the applicability of a plan to achieve a certain goal. Examples of preferences include an applicability heuristic for the whole plan, specification of forbidden of mandatory resources and the kind of applied strategy. An overview of different approaches to local adaptation of guidelines can also be found in [42]. In [43], a method called LASSIE using information extraction techniques is proposed for semi-automatic versioning of formalized computer-interpretable guidelines. However, none of these approaches seems amenable to efficient and scalable
industrial-strength implementations, which could cope with very large and ever growing guidelines repositories, with hundreds of versions and variants each.

2. Multi-version Clinical Guidelines

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. In this context, it is crucial to reconstruct the consolidated version\(^\text{2}\) 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 P at a time T on the basis of the clinical guideline versions which were valid at time T and applicable to the pathology of patient P. In other words, temporal concerns are important in the medical domain as in many other domains like law, accounting and scientific data analysis, and, thus, a guideline management system should be able to retrieve or reconstruct on demand any temporal version of a given clinical guideline to meet advanced application requirements.

Moreover, another kind of versioning, which we will call semantic versioning, plays a fundamental role, because clinical guidelines or some of their parts have limited applicability with respect, for instance, to the population of patients. In fact, a given guideline (e.g., involving treatment of heart diseases) may contain different recommendations which are not uniformly applicable to the same classes of patients: one general therapy may be non applicable to persons who suffer from some metabolic disorders (e.g., diabetes mellitus) or chronic diseases (e.g., kidney failure) or present some addictions (e.g., cocaine); one first-choice drug may not be administered to patients who are already under treatment with possibly interacting drugs (e.g., anticoagulants), or show genetic or acquired hypersensitivity or intolerance to some substances (e.g., patients with enzymatic defects or documented allergies), and so on. Hence, when dealing with a specific patient case, a physician may be interested in retrieving a personalized version\(^\text{3}\) of a clinical guideline, that is a version tailored to his/her use needs involving the patient's health state and anamnesis, and, thus, only containing recommendations which are safely and effectively applicable to the patient’s specific case.

\(^2\) The term “consolidated version” is borrowed from the legal field.

\(^3\) Notice that we use the term “personalized” as referred to the user of the computer system and not to the patient. Hence, in our examples, the target of personalization is the care provider using the guideline, whereas the temporal perspective, the diseases and other patient-specific information (e.g. medications he/she is taking), local settings etc. are the personalization coordinates.
In addition to linking guidelines to classes of patients, semantic versioning can also involve more generic applicability contexts (e.g., hospitals without PET diagnostic equipment or without a pediatric oncologist in the staff, or selected centers taking part to a clinical trial), which might require the application of a particular version of the general guideline, which may also no longer (or even not yet) be part of the consolidated state-of-the-art guideline. For instance, consider version v1 of a clinical guideline G which prescribes a biopsy to confirm a cancer diagnosis but has been superseded by a new version v2 which introduces a PET scan for the same cancer diagnosis, making in most cases the biopsy unnecessary. However, in some hospital H which is not equipped with a PET scanner, the right version of G to be followed is v1, although no longer considered valid by the medical community. Therefore, the applicable version of the guideline for context H is G(v1), with biopsy as a mandatory diagnostic means. This example also shows how temporal and limited applicability aspects may also interplay in the production and management of versions.

2.1. Temporal Versioning of Clinical Guidelines

As far as temporal versioning is concerned, several independent time dimensions are involved in the representation and management of clinical guidelines, in particular when we consider an environment also supporting the guideline authoring and approval process. Relevant time dimensions include:

- **Validity time.** It is the time (some part of) the guideline is considered in force by the medical community and, thus, is applied to patients. It has the same semantics of valid time as in temporal databases [44], since it represents the time the guideline actually belongs to the state-of-the-art of clinical practice in the real world.

- **Efficacy time.** Borrowing the term from the legal domain [33], it is the time (some part of) the guideline can be applied to a concrete case. It usually corresponds to validity, but it might be the case that an obsolete, superseded guideline continues to be applicable to a limited number of use cases. While such cases exist, the guideline continues its efficacy though no longer considered in force.

- **Transaction time.** It is the time (some part of) the guideline is stored in a computer system. It has the same semantics of transaction time in temporal databases [44].

- **Availability time.** It is the time (some parts of) the guideline becomes available to the information system, considering it made of the information flows and the human and computer resources that manage them [45]. In particular, it represents the time when users (e.g., physicians, nursing and technical staff) become aware of a new guideline (version).

Notice that, although they have a similar meaning, availability and transaction time are different and
must be modeled as independent time notions. In a similar vein, validity and efficacy time both have the semantics of valid time but represent orthogonal valid time notions. Both are necessary to correctly deal with cases as the one in the last described example: the guideline version G(v1) for the applicability context H can still be selected today as its efficacy includes current time, although its validity does not. On the other hand, transaction time [44] plays an important role when automatic management of information through computer systems is involved and, thus, should never be neglected, since it allows to keep track for audit purposes of the execution of retro- or pro-active modifications. For example, it might be the case that a physician makes a wrong decision in choosing a drug following the provisions of a guideline retrieved from the system when the returned consolidated version is actually out-of-date; the decision is taken while a modified version of the guideline (e.g., involving the adoption of some more effective and less potentially dangerous drug) is already available but has not been stored in the information system yet. Hence, transaction time is needed to ascertain a posteriori that the correct version was stored retroactively and, thus, the physician acted in good faith. In general, when clinical guidelines are routinely followed, multiple temporal dimensions are of crucial importance in the evaluation of medical malpractice for legal or insurance purposes.

Other time dimensions of interest have the nature of event time, that is the occurrence time of a real-world event that either initiates or terminates the valid time (validity or efficacy) of a (part of a) guideline [45]. They have the same semantics of event time as in temporal databases [46] and include:

- **Proposal time.** It is the time when a modification proposal to (a part of) an existing guideline is put forward by a general practitioner or developer [41].

- **Approval time.** It is the time when a proposed update to a (part of a) guideline is validated by a supervisor or team of supervisors before the original version can be modified [41].

- **Publication time.** Borrowing the term from the legal domain [33], it is the time a new guideline (version) is officially released by the issuing authority.

However, such event time coordinates can be considered as unchangeable properties of a (part of a) guideline version and, thus, they can be modeled as global (local) attributes rather than be used as further versioning dimensions. Moreover, considering an environment where only approved guidelines are stored, among those only publication time is relevant for queries and needs to be maintained. Availability time can also be viewed as the information system counterpart of the real-world event time [46].

Temporal versioning along multiple time dimensions can be plugged in clinical guidelines stored as XML documents in a repository by making temporal their XML encoding [47], that is introducing timestamps as annotations in the XML document.
2.2. Exploiting Ontological Information for Semantic Versioning of Clinical Guidelines

Semantic applicability of multi-version resources can be defined with reference to domain ontologies. Ontologies [48,49], which are conceptualizations of a domain into a machine-understandable format, have recently become quite popular with the advent of the Semantic web [50], where the introduction of common reference ontologies [51] is necessary to allow information and its interpretation to be shared by both human and automatic agents. Several semantic versioning coordinates, referencing specific domain ontologies, can be used to model patient-specific (e.g., diseases suffered or medications he/she is taking) or context-dependent applicability of guidelines (e.g., local constraints on resources or cultural differences). In this work, for simplicity and without loss of generality, we only consider patient diseases and demographic categorization as semantic versioning dimensions. Other dimensions can be added as orthogonal versioning dimensions and dealt with in a way similar to the ones introduced here.

![Diagram](image)

Fig. 1. Sample class hierarchies extracted from an ontology of diseases (a) and from a demographic ontology (b)

In particular, proper applicability of clinical guidelines to individual patients or homogeneous patient populations, with reference to their health state, can be defined according to a consensual taxonomy of diseases, like the ICD-10 endorsed by the World Health Organization [52] or the MeSH Section C maintained by the US National Library of Medicine [53]. Proper applicability of clinical guidelines with respect to the patient age and sex can be defined according to a consensual taxonomy of demographic classes, like the one present in the SNOMED-CT general ontology of medical terms [54]. In general, a medical ontology could be rather complex [55,56,57], including concepts, their relationships (e.g., IS-A or PART-OF links), properties/slots (possibly with restrictions), data types (defining the domain of some of the properties). In this respect, it does not differ from the ontologies usually considered for user
profiling (e.g., in [58]). However, the only ontological information which is relevant for semantic versioning and personalization of clinical guidelines, with respect to applicability to patient diseases, are the disease classes and their specialization/generalization relationships (because of the IS-A semantics, all the guidelines which are applicable to a class are also applicable to its subclasses), that is the disease class hierarchy. The same applies to other semantic versioning dimensions. Hence, in the following, we will only consider class hierarchies (viz., taxonomies) extracted from medical ontologies. A first sample fragment of such a hierarchy can be found in Fig. 1.a, which corresponds to a classification of principal heart diseases (taxonomy \(T_1\)). The second taxonomy \((T_2)\), that we will consider as a reference example, is the sample portion of demographic taxonomy shown in Fig. 1.b, which has been extracted from the SNOMED-CT ontology.

Before the personalization engine can be used to build a guideline version tailored to a specific use case, the patient must be classified with respect to the disease and demographic ontologies, either on the basis of electronic medical records by means of a suitable reasoning service [33], or through a profile explicitly supplied by the physician. In XML multi-version guideline repositories, reference to ontology concepts in a taxonomy can be added to the guideline representation and storage as a versioning coordinate [33]. In this way, applicability annotations can be embedded in the guideline documents to be used by automatic personalization tools. Obviously, the annotation of clinical guidelines that defines their semantic versioning must be effected and validated by medical domain experts, as part of the guideline drafting and approval process itself.

Notice that, in this work, we make the simplifying assumption that the reference ontology is constant and cannot be modified. A more realistic approach, where also reference ontologies are versioned as a consequence of updates (and the document semantic annotation scheme and personalization method take into account the versioning of ontologies), has been preliminary addressed for the legal domain in [59], and will be extended to clinical guideline management in our future work.

### 2.3. XML Encoding of a Multi-version Clinical Guideline

Since we aim at defining a generic multi-version encoding scheme which can be applied to any XML guideline format (e.g., those proposed in [13,14,15,16]), we will not introduce a specific DTD or XML-Schema for multi-version guidelines, but only show how the basic schema can be augmented in order to support versioning. Considering the DeGeL reference framework, our encoding scheme can be applied to the guideline representations at each of the three semi-structured, semi-formal and formal representation levels, being all XML documents. First, it can be applied at the semi-structured level: this is the target
level if clinical guidelines are stored and retrieved in textual form to be used as narrative guidelines. In this case, modifications are applied with an editor working at the semi-structured level, and then can be propagated preserving the multi-version encoding up to the formal level, so that multiple versions of guidelines are maintained both at the textual and at the so-called conceptual levels. To this end, the tools which are used to extract executable guidelines from the semi-structured level, can be made aware of the multi-version structure of the source documents to automatically produce multi-version guidelines at the formal level. Second, the multi-version encoding can directly be applied at the (semi-)formal level, where guidelines are stored and retrieved by end users in their executable form. At this level, guidelines can be represented and stored in an XML format encoding a task network or flowchart (e.g., even a “standard” XML-based workflow specification language, like WS-BPEL [60] or XPDL [61], could be used to this purpose), to which the multi-version encoding scheme can be applied. In this way, modifications which produce new versions are applied to the guideline conceptual model, via an editor working at the (semi-)formal level; the initial textual format is no longer maintained, as modifications cannot be propagated back to narrative guidelines, which could be a disadvantage.

In the following, we introduce and explain the proposed encoding scheme. Just in order to make it a little easier to follow, the proposed versioning examples concern a narrative guideline. Starting from the non versioned structure of the XML guideline document, that is from the snapshot schema [62], the i-th element “<element-i> contents of i-th element </element-i>” can be versioned by introducing suitable <version> sub-elements to mark and delimit the boundaries of versions of its contents as follows: “<element-i> <version number="1"> 1-st version of contents of i-th element </version> <version number="2"> 2-nd version of contents of i-th element </version> … <version number="N"> N-th version of contents of i-th element </version> </element-i>.” Each <version> element will contain a <pertinence> sub-element which is used to assign the versioning coordinates, that is timestamps and semantic annotations, to the container version (we assume “version” and “pertinence” were not already used as element names in the snapshot schema; otherwise, a suitable namespace [63] for versioning can be defined and referenced to disambiguate them). The proposed encoding scheme is independent from any specific snapshot schema and general enough to allow the adoption of an arbitrary number of temporal and semantic dimensions as versioning coordinates.

Let us now consider a concrete example of temporally and semantically versioned clinical guideline. For the sake of simplicity, but without loss of generality, we only consider in the examples which follow one time dimension (i.e., validity) and two semantic dimensions (i.e., reference to classes in reference taxonomies like the ones in Fig. 1). Fig. 2 shows a fragment of the structure of a sample multi-version
guideline involving recommendations for the treatment of unstable angina patients. The figure displays the text organization, which has a three-level section structure, where section 3.2 has two different versions, namely 3.2(v1) and 3.2(v2), and section 3.2(v1).2 has four different versions, namely 3.2(v1).2(v1), 3.2(v1).2(v2), 3.2(v1).2(v3) and 3.2(v1).2(v4). The multi-version XML encoding of such guideline is shown in Fig. 3.

**RECOMMENDATIONS**
1. IDENTIFICATION OF PATIENTS WITH RISK OF UNSTABLE ANGINA
   ...
2. INITIAL EVALUATION AND MANAGEMENT
   ...
3. EARLY HOSPITAL CARE
   3.1. Initial Treatment Strategy
   ...
   3.2. Drug Therapy
   3.2(v1). Anti-Ischemic and Analgesic Therapy
      3.2(v1).1. Therapy with nitrates
      ...
      3.2(v1).2. Therapy with beta-blockers
      3.2(v1).2(v1). administration of drug D1...
      3.2(v1).2(v2). administration of drug D2...
      3.2(v1).2(v3). administration of drug D3...
      3.2(v1).2(v4). administration of drug D3'...
      ...
      3.2(v1).3. Therapy with ACE inhibitors
      ...
      3.2(v2). Antiplatelet/Anticoagulant Therapy
   ...
4. CORONARY REVASCULARIZATION
   ...
5. LATE HOSPITAL CARE
   ...

Fig. 2. The structure of a fragment of a sample multi-version clinical guideline.

Within a `<pertinence>` element, `<valid>` and `<applies>` elements are used to assign the temporal and semantic pertinence, respectively, to the version which contains it. In particular, in order to define the applicability of guideline parts, we introduce references to the taxonomy classes by means of the numbering scheme corresponding to the pre-order visit of a tree, which allows nodes in the reference taxonomy to be ranked and univocally identified.

Generally speaking, the semantic pertinence of a node references one or more classes selected from one or more reference taxonomies. In order to distinguish between the different taxonomies used to annotate a collection of XML guideline documents, we assume that each of them is identified through a unique name $T_i$ and, thus, each of its classes can be uniquely denoted by means of the pre-order rank $j$ of the class in the taxonomy as $T_i, C_j$. 
Fig. 3. A fragment of the guideline of Fig. 2 with multi-version encoding and applicability annotations

The taxonomy class numbering scheme is also extended with post-order numbers for query processing purposes, as the pre-order and post-order visit properties of trees allows us to quickly check ancestor-
descendant relationships between the classes [64]. This aspect is particularly important because, when a portion of a guideline is applicable to a class, it is also applicable to all its descendant classes. This means that, in order to reconstruct the appropriate version of all and only document portions which are applicable to the class \( T_i.C_j \), it is necessary to consider not only \( T_i.C_j \) but also \( T_i.C_j \)'s ancestors. For instance, with reference to the disease taxonomy \( T_i \), the portions of the guideline which are relevant for a patient affected by unstable angina \( (T_i.C_5) \) are those which apply not only to \( T_i.C_5 \) but also to \( T_i.C_4, T_i.C_3 \) and \( T_i.C_1 \). The combination of the two codes are represented in the upper left corner of the taxonomy classes in Fig. 1, in the form: (pre-order, post-order), with pre-order highlighted in boldface. For example, in \( T_i \), the class “Myocardial ischemia” has pre-order “3.”, which is also its identifier, whereas its post-order is “6.” Hence, checking the ancestor-descendant relationship between nodes is reduced to a simple comparison between integer values: for example, the descendant classes of the class “Myocardial ischemia” are those classes that have pre-order rank > 3 and post-order rank < 6 (i.e., region \( D \) in Fig.4), whereas the ancestor classes have pre-order rank < 3 and post-order rank > 6 (i.e., region \( A \) in Fig.4).

Validity and applicability properties are inherited by descendant nodes in the XML tree-structure unless locally redefined with a new version definition. Therefore, there is no reason to repeat the valid or applies annotation when the pertinence is not changed from the ancestor version in the XML tree-structure. In general, redefinition may involve only a subset of the versioning dimensions, while the others dimensions are inherited.

![Fig. 4. Classes represented in terms of their pre-order and post-order numbers (taxonomy \( T_i \)).](image)
With reference also to Fig. 2, the XML fragment in Fig. 3 shows, within the outermost <recommendations> element, a hierarchical structure based on three levels of sections, with the <foSection>, <soSection> and <toSection> elements denoting first-, second- and third-order sections, respectively. The <recommendations> element is composed of one version, which defines its global semantic and temporal pertinence, that is applicable to classes $T_1.C_3$ and $T_2.C_I$ with respect to the taxonomies in Fig. 1 (adult patients with myocardial ischemia, respectively) and valid from 1980 on (a “9999-99-99” endpoint is used to represent a right-unlimited interval). It is made of several first-order sections (see also Fig. 2), of which only section 3 is evidenced in the Figure. Such a section, made of only one version to specify applicability to taxonomy class $T_1.C_4$ (adult patients with angina pectoris), deals with “Early Hospital Care.” Its temporal pertinence is inherited from the container element.

In general, by means of redifinitions, we can introduce complex validity and applicability properties including extensions or restrictions with respect to ancestors for each part of a document. For instance, considering the taxonomy $T_1$, the applicability assignment to section 3, which we just described, is a restriction and the attribute to is used to this end. Actually, the applicability assigned to the version is the intersection of the to value and of the value inherited by the ancestor version (in this case, $T_1.C_4 \cap T_1.C_3$, which equals class $T_1.C_4$ since it is a subclass of $T_1.C_3$). The same applies to second-order section 3.2 (entitled “Drug Therapy”), whose first version (entitled “Anti-ischemic and Analgesic Therapy”) applies to class $T_1.C_5$ (unstable angina), which is also a restriction, whereas the second version (entitled “Antiplatelet/Anticoagulant Therapy”) is also applicable to class $T_1.C_7$ (myocardial infarction), which is an extension indeed. The attribute also is used in this case, and the applicability assigned to the version is the union of the also value and of the value inherited by the ancestor version (class $T_1.C_4 \cup T_1.C_7$). In other words, the contents of section 3.2(v2) apply both to angina pectoris and myocardial infarction (adult) patients.

The third-order section 3.2(v1).2 entitled “Therapy with Beta-blockers” is made of several versions, each one dealing with the administration of a specific drug and having its own temporal pertinence, whereas the (inherited) applicability is the same (namely class $T_1.C_5$, unstable angina) with respect to taxonomy $T_1$; with respect to taxonomy $T_2$, all versions from 1 to 3 inherit their applicability (namely class $T_2.C_I$, adult); whereas the applicability of the fourth version has been redefined as class $T_2.C_I \cap (T_2.C_4 \cup T_2.C_5) = T_2.C_4 \cup T_2.C_5$ and, thus, such version applies to pregnant women and elderly persons.

In order to derive the validities of the four drugs shown in the Figure, we assume the recommendations underwent the evolution which follows. Drug D1 was introduced in 1980 and then replaced by the drug D2 in 1991. However, the use of drug D2 was suspended from 1999 to 2000, period during which it had
been under investigation since suspected of causing adverse reactions. In 2004, due to evidence of long-term adverse effects, D2 was definitely withdrawn. Drug D3 has been introduced in 1985 along with a mild version D3′, that represents the elective choice when it has to be administered to pregnant women and elderly persons. Hence, the resulting history of recommended beta-blockers according to the guideline in Fig. 2 (which will in fact correspond to the answers to a sequence of snapshot queries issued on the multi-version document) is the following:

- from 1980 to 1984: drug D1
- from 1985 to 1990: drugs D1, D3 and D3′
- from 1991 to 1998: drugs D2, D3 and D3′
- from 1999 to 2000: drug D3 and D3′
- from 2001 to 2003: drugs D2, D3 and D3′
- from 2004 on: drugs D3 and D3′

As for 3.2(v1).2(v2) in the Figure, versions can be assigned multiple intervals as validity: this corresponds to adopting *temporal elements* [44], that is disjoint union of intervals, as timestamps.

### 2.4. Modification of a Multi-version XML Guideline

The first multi-version XML encoding of a guideline needs to be created from scratch. The encoder must manually assign the right semantic pertinence to the different text portions, possibly helped by natural language processing tools which highlight the linking between relevant words or sentences and the reference taxonomy classes (usually, the temporal pertinence is initially the same for all the guideline parts, i.e., valid from the release date of the guideline). The multi-version XML encoding can then easily be generated in automatic way from this assignment. To this purpose, no special method nor special tool is needed, as any existing ontology-based guideline markup tool can be used (e.g., this encoding phase is basically a subtask of the derivation of the semi-structured and semi-formal guideline representations in the DeGeL environment). After the first guideline encoding has been generated, further temporal and semantic versions are automatically added to the multi-version encoding when changes are applied.
Fig. 5. Management of modifications in a traditional versus multi-version setting. The modification producing the XML fragment (b) from (a) in a traditional setting, produces the XML fragment (c) in the multi-version setting.

For instance, let us consider a (narrative or executable as well) XML-encoded guideline created on 2010-01-01 and containing an `<element_X>` XML element, whose contents have been linked by the author to a class XY of a reference taxonomy $T_1$. If the guideline is modified, with a suitable editor, on 2011-11-01 by changing the (textual or algorithmic as well) contents of `<element_X>` from “OLD CONTENTS” to “NEW CONTENTS” and linking it also to a class YZ of another reference taxonomy $T_2$.

In a traditional setting, the editor used for the modification simply overwrites in the XML file the old XML encoding shown in Fig. 5(a) of `<element_X>`, by replacing it with the new encoding shown in Fig 5(b). In a multi-version setting, the editor used for the modification (which could even have the same user interface and exposed functionalities of the traditional one) has to replace the old encoding shown in Fig. 5(a) of `<element_X>` with the one shown in Fig. 5(c). Basically, in this simplest scenario, we can say that the traditional and multi-version authoring environments only differ in the way the editor saves the modified XML files. In such a scenario, guideline authors could even be unaware of their multi-version nature. Therefore, the solutions which have been devised for complex guideline formalization and maintenance problems (e.g., merging of formal guideline versions independently derived from the same origin document as studied by Peleg and Kantor in [40] or automated derivation of a new formal representation after a revision in the origin narrative guideline as studied by Kaiser and Miksch in [43]) are still viable and can be implemented in the same way also in a multi-version setting. Once the modifications that need to be applied to an XML-encoded guideline to solve such complex problems are well defined (i.e., exactly the same modifications required in a traditional setting), we simply have to apply them as shown above to produce a multi-version representation. Notice that such modifications can
also be derived in an automated way by comparing the versions which have to be merged via some XML difference search method (e.g., [65]).

However, in a more complex scenario, new functionalities involving the multi-version structure (e.g., to change the temporal pertinence of a guideline portion without changing its contents) could also be exposed to guideline authors by advanced editors. Hence, we briefly describe the modifications which are required to the user interface of an advanced guideline graphical editor to maintain an XML multi-version guideline after its creation. In order to apply changes to a guideline, the portion to be modified must be individuated as a sub-tree rooted on a selected element within the XML document. The selection could be easily effected by means of a graphical editor like the DeGeL URUZ tool, also showing the inner tree structure of the document. The user interface must be extended to also display the temporal and semantic pertinence of the selected node (version). Once the guideline portion to be modified has been individuated, there are two basic operations which can be applied: modification of contents (e.g., to revise contents or to add a new subsection in a narrative guideline) or update of the temporal and semantic pertinence (e.g., to extend the applicability of a prescription to a new disease or to revive a superseded recommendation by assigning a new validity). The new contents or the new pertinence, respectively, must be supplied by the user via the editor interface. Once his/her work is completed, updates are applied to the document via the execution of modification primitives, aware of the multi-version document structure, which must be embedded in the editor functionalities. The semantics of such primitive operations is basically the same that has been defined in [32] for the maintenance of temporal XML norm documents in the legal domain. Their definitions in [32] can easily be extended to also take into account the semantic versioning aspects. In particular, their execution guarantees the respect of the multi-version encoding scheme of Sec. 2.3 and that integrity constraints concerning temporal and semantic versioning are preserved. Complex changes, involving non-local modifications of the document tree structure (e.g., moving a sub-tree or merging two sub-trees), must be decomposed and mapped onto a sequence of primitive modifications. However, the classification of complex changes, which can be made available to guideline authors as advanced editor functions, and the study of their mapping onto primitives is beyond the scope of this work and will be dealt with in future research.

As it can also be guessed from the example in Fig. 5, since modifications always add new material to XML files, the real problem of managing multi-version guidelines is that their size grows very rapidly as dozens or hundreds of modifications are applied. The growth rate can be so high that, in a large guideline repository, the traditional database technology is bound to fail soon in searching them and scalability becomes the main issue.
2.5. Query Specification for Extracting Personalized Guideline Versions

Repositories of clinical guidelines in narrative form, like the US National Guideline Clearinghouse [7] or the UK National Library of Guidelines [8], are usually managed by traditional information retrieval systems where users are allowed to access their contents by means of keyword-based queries expressing the subjects they are interested in. By adopting a clinical guideline encoding like the one presented in Section 3, users are further offered the possibility of expressing temporal and semantic specifications to build a personalized version of the retrieved guideline. The same personalization facilities can be embedded in a repository of clinical guidelines in (XML-encoded) executable format.

In particular, the queries can contain four types of constraints: temporal, structural, textual and applicability. The constraints can be explicitly specified by the user through a suitable interface or automatically derived from the query execution context. The four types of constraints are completely orthogonal and enable a full support of multi-dimensional selection and personalization.

Let us introduce an example and focus first on the applicability constraint. Consider again the taxonomy $T_1$ in Fig. 1 and the guideline fragment in Fig. 2: for the treatment of John Smith, who we assume to be an “infarctuated” patient (i.e., belonging to class $T_1.C_7$), the sample recommendations in Fig. 2 will be selected as pertinent, but only the second version of section 3.2 will be actually presented as applicable.

Furthermore, the applicability constraint can be combined with the other three ones in order to fully support a multi-dimensional retrieval. For instance, a physician (or an health insurance officer) could be interested in all the guidelines ...

- ... which have a second-order section whose title (structural constraint) contains the word anticoagulant (textual constraint), ...
- ... which were valid between 2007 and 2008 (temporal constraint), ...
- ... and which are applicable to an adult patient suffering from unstable angina (applicability constraint).

His/her request can be expressed as personalization query [24], that is an XQuery statement having the standard FLWR syntax [66] as follows:

```
FOR $a IN guidelines.xml
WHERE textConstr('a//soSection/title/text(), anticoagulant')
AND tempConstr('vTime OVERLAPS PERIOD(2007-01-01,2008-12-31)')
AND applConstr('T1.C5') AND applConstr('T2.C1')
RETURN $a
```
In general, the WHERE clause can contain the conjunction of a textual, a temporal and an applicability clause, as in the example. Such selection clauses are expressed by means of the Boolean functions textConstr, tempConstr and applConstr, which can be used to specify textual, temporal and applicability constraints, respectively.

The textual clause contains a single occurrence of the textConstr function, where the first argument is a node-labeled twig pattern, defined on the snapshot schema of the XML database in XPath syntax [67], which must match the string passed as second argument of the function. It is used to express a pattern of selection predicates on multiple elements having some specified tree-structured relationships and defines the portion of interest in each state of the XML guideline documents contained in the database.

```
<recommendations>
  ...
  <foSection number="3">
    <title>Early Hospital Care</title>
    ...
  </foSection>
  ...
  <foSection number="2">
    <title>Antiplatelet/Anticoagulant Therapy</title>
    ...
  </foSection>
  ...
</recommendations>
```

Fig. 6. The guideline fragment returned by the personalization query of the example.

The temporal clause can contain a conjunction of one occurrence of the tempConstr function for each of the supported temporal dimensions, which is used to express the temporal conditions the versions of interest must satisfy. The argument of the tempConstr function is a string encoding a temporal selection predicate with a syntax similar to the one adopted for the TSQL2 temporal query language [68]. More specifically, the available temporal predicates are OVERLAPS, which is used to execute a time-slice query (i.e., to retrieve a temporally consistent set of consecutive versions valid in a given time period), and CONTAINS, which is used to execute a snapshot query (i.e., to retrieve a single temporal version valid at a given time point). Notice that snapshot queries are dealt with as a particular case of time slice query over a period of unit length (e.g., ‘CONTAINS 2010-06-01’ is equivalent to ‘OVERLAPS PERIOD (2010-06-01, 2010-06-01)’).

Finally, the applicability clause is a Boolean combination (with AND and OR operators) of elementary
applicability predicates. Each applicability predicate is expressed by means of the \texttt{applConstr} function, involves an applicability constraint on one semantic dimension and can appear in positive or negated (i.e., preceded by the \texttt{NOT} operator) form. The argument of the \texttt{applConstr} function is a string with the general form '\texttt{Ti.Cj:depth}', where \texttt{Ti} is the reference taxonomy, \texttt{Cj} is the class the user is interested in and \texttt{depth} is an optional numeric parameter. It is worth noting that as a consequence of the inheritance semantics of applicability, with reference to the disease taxonomy, if a user is looking for guidelines concerning say unstable angina (class \texttt{T1.C5}), then the returned recommendations can also be applicable to angina pectoris (class \texttt{T1.C4}), myocardial ischemia (class \texttt{T1.C3}) and heart disease (class \texttt{T1.C1}). In such a case, by using the optional \texttt{depth} parameter, the user is able to limit the applicability scope to the requested class ancestors located up to \texttt{depth} steps upward in the class hierarchy. For instance, \texttt{applConstr('T1.C5:0')} limits the applicability search to the class \texttt{T1.C5} only, whereas \texttt{applConstr('T1.C5:1')} has to be used to retrieve guidelines applicable to classes \texttt{T1.C5} and \texttt{T1.C4} (which is one step over \texttt{T1.C5}) in the disease taxonomy \texttt{T1}.

3. Efficient Personalized Access To Multi-Version XML Clinical Guidelines

In order to efficiently support the complex requirements of an XML multi-version clinical guidelines personalization engine, the underlying data management infrastructure has to be carefully devised. To this end, two alternative solutions exist. One option is to rely on traditional \textit{off-the-shelf XML engines}, offering intrinsic XML data storage and management facilities. In this case, multi-version guidelines are dealt with as standard XML documents and, thus, stored in the XML repository using the data structures made available by the engine. The chosen storage granularity is typically the whole document, although different options could be available. However, the problem is that those engines are not aware of the temporal and semantic versioning aspects of the managed data. As a consequence, a software stratum has to be built on top for handling the additional features, and query optimization and indexing techniques especially suited for multi-version XML documents are very difficult, if not impossible, to apply. This introduces large overheads in performing temporal and applicability filtering on the retrieved data in order to return only the XML guideline portions satisfying all the user constraints. The second design option is to build a \textit{native multi-version XML query processor}, which is able to index the XML guideline repository and provide all the required facilities for a personalized access through the introduction of a suitable temporal and semantic \textit{slicing operator}, aware of the multi-dimensionality of data and of the query structure. Extending to the multi-version setting the term coined for temporal data [44,69], \textit{slicing} consists of selecting the qualifying data while retaining a consistent multidimensional timestamp and semantic
This section provides a detailed description of the design of a personalization engine based on the native solution by showing how some of the best performing technologies available for XML data management can be extended and optimally combined. In particular, step by step, we will spot the changes that must be applied to a conventional XML pattern matching engine to incrementally transform it into a multidimensional personalization engine. The advantage of this approach is that the processor under construction benefits from the XML pattern matching techniques proposed in the literature, where the focus is on the structural aspects which are intrinsic also to multi-version XML documents, and, at the same time, can be freely extended to become temporally and semantically aware.

We begin by providing some background in Section 3.1 and 3.2, where the multi-version guideline slicing problem is defined. Then, we discuss the foundations of a native XML personalization engine, with particular attention to the underlying indexing and main memory structures (Section 3.3). In Section 3.4 a temporal and semantic indexing scheme is presented, which extends the inverted list technique proposed in [70], in order to allow the storage of multi-version XML guidelines. Finally, we introduce a flexible technology supporting multi-version guideline slicing (Sections 3.5 and 3.6). It consists of alternative solutions supporting slicing on the adopted storage scheme, all relying on the holistic twig join technology [71], which is one of the most popular approaches for XML pattern matching. The proposed solutions act at the different levels of the holistic twig join architecture, aiming at limiting main memory space requirements, I/O and CPU costs. They include the introduction of novel algorithms and the combined exploitation of different access methods.

3.1. Preliminaries

In the following, we consider a multi-version XML clinical guidelines database (MVXMLdb) as a collection of XML guideline documents conforming to the multi-version data model introduced in section 2. Each document $D^{MV}$ in the collection can be represented as an ordered labeled tree where element nodes are versioned along either the temporal dimensions, or semantic dimensions, or both. Being $n^{MV}$ a (versioned) node in a multi-version XML guideline document, we denote by $\text{lifetime}(n^{MV})$ its timestamp, and by $\text{applicability}(n^{MV})$ its semantic annotation.

In general, the timestamp is a multidimensional temporal element defined as the Cartesian product of a (open to the right) time interval for each of the supported temporal dimensions. Since any version is potentially subject to future changes with respect to all the supported time dimensions, we will adopt the symbol “UC” (Until Changed [44]) as endpoint to represent some data which has not been changed yet.
(“9999-99-99” in the XML encoding). For instance, if we consider the three temporal dimensions which are the most useful for the medical domain, that is validity time \((vt)\), efficacy time \((et)\), and transaction time \((tt)\), and assume that the time granularity is the day, the temporal pertinence of a node representing a version of a guideline section could be \([1990-01-01, 1995-07-31]_vt \times [1990-01-01, UC]_{et} \times [1990-02-15, 2000-04-01]_tt \cup [1997-06-10, UC]_{et} \times [1997-06-10, UC]_{et} \times [1996-05-10, UC]_{tt}\). It is worth noting that the adoption of timestamps made up of temporal elements instead of convex multidimensional intervals avoids the duplication of version contents in the presence of a temporal pertinence with a complex shape.

Semantic annotations are represented as a Boolean formula involving applicability classes selected from the considered taxonomies. Specifically, the semantic pertinence “applicable to \(\bigcap_{i} \left( U_j \ T_i \ C_j \right) \)” of a node \(n^{MV}\) translates into the following applicability formula: \(applicability(n^{MV}) = \Lambda \left( V_j T_i \ C_j \right)\).

Fig. 7. Internal representation and slicing example of the XML guideline fragment in Fig. 3

For instance, the top part of Fig. 7 depicts the internal tree structure of a fragment of the XML guideline shown in Fig. 3. For ease of presentation, timestamps are defined with the granularity of a year. Applicability contexts refer to classes in the taxonomies shown in Fig. 1.
3.2. The Multi-version Slice Operator

Personalization queries like those considered in section 2.5 are implemented by means of a multi-version slice operator:

\[
mv\text{-}\text{slice}(\text{twig}, \text{t-window}, \text{a-formula})
\]

which is the main underpinning of the native multi-version query engine. The three arguments correspond to the structural/textual constraints, temporal constraints and applicability constraints, respectively, which users can express in the personalization query. In particular, \text{t-window} is the Cartesian product of an interval for each of the supported time dimensions, which derive from the translation of the temporal constraints specified as argument of the \text{tempConstr} predicates.

On the other hand, \text{a-formula} is a logic formula \((\neg \mathcal{H}(T_t,C_i;\text{depth}_t) \lor T_t,C_j;\text{depth}_t) \land \cdots \lor \neg \mathcal{H}(T_t,C_n;\text{depth}_t))\), which derives from the translation of the semantic constraints specified by the applicability clause in the user query. Each \(\neg \mathcal{H}(T_t,C_i;\text{depth}_t)\) term corresponds to the translation of a \text{applConstr}(T_t,C_i;\text{depth}_t) predicate, where the symbol \(\neg\) is the negation sign \(\neg\) if the corresponding predicate was in negated form (\(\neg\) is null otherwise). More precisely, the function \(\mathcal{H}\) unrolls a taxonomy navigational pattern into applicability to the disjunction of the classes involved (e.g., \(\mathcal{H}(T_1.C_5:1) = T_1.C_5 \lor T_1.C_4\)). For instance, the translation of the applicability clause \(\text{applConstr}(T_1.C_2:0) \lor \text{applConstr}(T_1.C_5:1) \land \neg \text{applConstr}(T_2.C_2:0)\)” gives rise to \text{a-formula} = \(T_1.C_2 \land (T_1.C_5 \land T_1.C_4 \land \neg T_2.C_2\).

When \text{tempConstr} or \text{applConstr} parameters are omitted in the query, default conditions for \text{t-window} and \text{a-formula} are assumed in order to match any possible version: the whole time domain for the former and the Boolean value \(\text{TRUE}\) for the latter.

The slice operator \(mv\text{-}\text{slice}(\text{twig},\text{t-window},\text{a-formula})\) simultaneously retrieves the portion of each version of the multi-version XML guidelines in the database \(MVXMLdb\) which matches the given XML query twig pattern \text{twig}, overlaps the given time period \text{t-window}, and is applicable to any class in the given portions of the reference taxonomies as specified in \text{a-formula}. The results are combined back into a period-stamped and semantic-annotated representation.

A slicing example of the multi-version XML document in the top part of Fig. 7 is shown in the bottom part of the same figure. The personalization query asks for all third-order sections descendant of the first-order section numbered “3” (path: \text{foSection[@number=“3”]//toSection}) which were valid in the period [2007,2009] and which are applicable to men (or, more generically, adults) affected by unstable
angina (or angina pectoris), that is matching the a-formula \( (T_1.C_5 \lor T_1.C_4) \land (T_2.C_2 \lor T_2.C_i) \). The outcome is represented by the two portions of the document which satisfy all the constraints, where each qualifying portion is denoted as slice.

Formally, a multiversion slice is a mapping from nodes in twig to nodes in MVXMLdb, such that:

(i) query node predicates are satisfied by the corresponding document nodes and determine a tuple \((n_1^{MV}, \ldots, n_k^{MV})\) of all the nodes in MVXMLdb that produce a distinct match with twig;

(ii) \((n_1^{MV}, \ldots, n_k^{MV})\) is structurally consistent, i.e., all the \(n_i^{MV}\) nodes belong to the same multiversion XML document and the parent-child and ancestor-descendant relationships between query nodes are satisfied by the corresponding document nodes;

(iii) \((n_1^{MV}, \ldots, n_k^{MV})\) is temporally consistent, i.e., the intersection of the lifetimes of all nodes \(T = \text{lifetime}(n_1^{MV}) \cap \ldots \cap \text{lifetime}(n_k^{MV})\) (which is not empty as the tuple is structurally consistent) overlaps the given time period: \(T \cap t\text{-window} \neq \emptyset\);

(iv) \((n_1^{MV}, \ldots, n_k^{MV})\) is semantically consistent, i.e., considering the applicability annotation of a node \(n_i^{MV}\) expressed in Disjunctive Normal Form (DNF) \(\text{applicability}(n_i^{MV}) = \bigvee_j \text{appl}_j(n_i^{MV})\), at least one of the disjuncts in the DNF applicability annotation of each node \(n_i^{MV}\) logically implies the applicability constraints formula [36]: for each \(i\), exists \(j\) such that \(\text{appl}_j(n_i^{MV}) \Rightarrow \text{a-formula}\).

For instance, in the reference example, the tuple (B,D) is structurally consistent being B a foSection node with the number attribute equal to 3, D a toSection node and D a descendant of B, and is also semantically consistent as \(\text{applicability}(D) = T_1.C_5 \land T_2.C_i \Rightarrow (T_1.C_5 \lor T_2.C_i) \land (T_2.C_2 \lor T_2.C_i)\). However, it is not temporally consistent as \(\text{lifetime}(B) \cap \text{lifetime}(D) = [1980,1990] \cap [1980,1990] \cap [2007,2009] = \emptyset\).

Notice that, since the applicability annotation of a node has been defined as \(\text{applicability}(n^{MV}) = \bigwedge_i \bigvee_j \text{T}_i.C_j\), its DNF can easily be computed as \(\bigvee_j \bigwedge_i \text{T}_i.C_j\) and, thus, \(\text{appl}_j(n^{MV}) = \bigwedge_i \text{T}_i.C_j\).

### 3.3. Foundations of Native XML Clinical Guidelines Query Processor

Finding all occurrences of a query twig in a very large XML database is a core operation needed for the execution of personalization queries, which goes beyond the common query capabilities of a relational DBMS. In order to deal with the twig matching problem, a lot of work has been done on XML query processing techniques (see, e.g., [70,71,72,73,74,75]). They show that capturing the XML document structure using traditional indices is a good solution, relying on which it is possible to devise efficient structural or containment join algorithms for twig pattern matching. Early proposals in this context (e.g.,
[70]) were based on the decomposition of the query into a set of binary (parent-child and ancestor-descendant) relationships between pairs of nodes. In this way, the twig query pattern can be matched by testing each of the binary relationships against the XML database and “stitching” the resulting basic matches together. The main limitation of these approaches is that they suffer from very large intermediate result size, even when the input and the final result sizes are much more manageable. To address the problem, Bruno, Koudas and Srivastava proposed in [71] a holistic twig join for matching XML query twig patterns. The holistic twig join approach stores XML data by using the same indexing scheme as proposed in [70], but a chain of linked stacks is then used to compactly represent partial results of individual query root-to-leaf paths. Finally, efficient algorithms merge the sorted lists of participating element sets together and, in this way, avoid creating large intermediate results. In order to understand the foundations of a holistic XML query processor, in the rest of this section we will analyze these aspects in detail, while in the next section we will describe how to seamlessly extend such an architecture in the direction of temporal and semantic versioning management.

3.3.1. Position-based Indexing Scheme

The indexing solution for XML data proposed in [70] include:

1. a numbering scheme that encodes each element and string occurrence by its position within the tree structure of the XML document to which it belongs;
2. an extension of the classic inverted index used for information retrieval which maps each element or string to the inverted list of its occurrences in the XML database.

The position of a string occurrence in the XML database is represented as a tuple (DocId, LeftPos, LevelNum) in each inverted list and, analogously, the position of an element occurrence as a tuple (DocId, LeftPos:RightPos, LevelNum) where (a) DocId is the identifier of the document, (b) LeftPos and RightPos are the word counts from the beginning of the document to the start and end of the element, respectively, and (c) LevelNum is the depth of the node in the document structure.

In this context, structural relationships between tree nodes can be easily determined:

- **ancestor-descendant:** A tree node \( n_2 \) encoded as \((D_2,L_2:R_2,N_2)\) is a descendant of the tree node \( n_1 \) encoded as \((D_1,L_1:R_1,N_1)\) iff \( D_1 = D_2, \) \( L_1 < L_2, \) and \( R_2 < R_1; \)
- **parent-child:** \( n_2 \) is a child of \( n_1 \) iff it is a descendant of \( n_1 \) and \( L_2 = L_1 + 1. \)
3.3.2. The Holistic Twig Join Approach

The holistic twig join approach [71] consists of stack-based algorithms which adopt the four level architecture depicted in Fig. 8. In particular, both the algorithms presented in [71], the former for path matching and the latter for twig matching, essentially work on paths. In case of path matching, the path is the query itself (e.g., //recommendations//toSection), whereas, in the other case, the algorithm works on each individual root-to-leaf query path.

Given a path q₁, ..., qₙ, involving n nodes, for each node qᵢ, there is an inverted index Iᵢ (level L0 in Figure), which is associated with the data elements or strings that match the query predicate at node qᵢ. In particular, it contains the positional representation of all the matching database nodes. For instance, given the path //foSection//soSection/toSection and the data tree shown in Fig. 7 (ignoring versioning annotations), the inverted indices are I₇foSection, I₇soSection, and I₇toSection, where I₇foSection contains one tuple encoding node B, I₇soSection contains two tuples encoding nodes C and H, and I₇toSection contains five tuples encoding nodes D, E, F, G and I. Moreover, the algorithms associate qᵢ to a stack Sᵢ (level L2 in Figure). At each step of the algorithms, such a stack contains data nodes from Iᵢ. Each data node in the stack consists of a pair (positional representation of a node from Iᵢ, pointer to a node in S[parents(Iᵢ)])). In particular, at every computation step, the data nodes in Sᵢ are guaranteed to lie on a root-to-leaf path in the XML database and the set of stacks represent in linear space a compact encoding of partial and total answers to the query twig pattern. The skeleton of the two holistic twig join algorithms (HTJ algorithms in the following) is presented in Fig. 9.

![Diagram of the basic four level holistic twig join architecture](image-url)
At each iteration, the algorithms identify the next node to be processed. To this end, for each query node \( q_i \), level L1 contains in the main memory buffer \( B_{qi} \) the node in the inverted index \( I_{qi} \) with the smallest LeftPos value and not yet processed. Among those, step 1 chooses the node with the smallest value, let it be \( n_q \). Then, given the knowledge of such node, the algorithms remove partial answers from the stacks that cannot be extended to total answers and push the node \( n_q \) into the stack \( S_q \). Whenever \( n_q \) matches a leaf node of the query path and, thus, it is pushed into a “leaf” stack, the set of stacks contains an encoding of total answers. Therefore, step 4 outputs these answers at SOL Level and immediately deletes \( n_q \). For instance, given the query //foSection//soSection//toSection, let us assume that the next node to be processed is H, that is nodes B, C, D, E, F and G have already been selected from level L1. At this step, \( S_{foSection} \) and \( S_{soSection} \) contain B and C, respectively, while \( S_{toSection} \) is empty. In such a case, the partial answer (B,C) cannot be further extended because H belongs to a root-to-leaf path different from that of C and, thus, C is deleted from \( S_{soSection} \). Node B instead is maintained in \( S_{foSection} \) because (B,H) can be extended to a total answer, as it is checked when processing I.

The base algorithms sketched in [71] have been further improved by [72,73,75].

### 3.4. Indexing Scheme Extensions for Multi-version Guideline Management

Since multi-version XML clinical guidelines are XML documents, regardless of the meaning of temporal and semantic annotations, they can be indexed using the interval-based scheme described above. Although timestamps and applicability attributes can be indexed as “standard” tuples, timestamped and annotated nodes have a specific semantics which should be exploited when guideline documents are accessed and, in particular, when the slice operator is applied. The idea is to add time and applicability to the interval-based indexing scheme by substituting the inverted indices of [70] with multi-version inverted indices. In each multi-version inverted index, in addition to the explicit attributes encoding the position of an element occurrence in the multi-version XML database, the tuple

\[(DocId, \text{LeftPos:RightPos}, \text{LevelNum | SemAppl, TempPer})\]

also contains two implicit attributes: \text{SemAppl} and \text{TempPer}. 

Fig. 9. Skeleton of the holistic twig join algorithms (HTJ algorithms).
The attribute TempPer represents a single multi-dimensional period belonging to the temporal pertinence of the tuple, whose value is expressed via a list of From:To pairs, one for each involved temporal dimension. All the multi-version inverted indices are defined on the same temporal domain, so that tuples coming from different inverted indices are always comparable from a temporal point of view. Therefore, given the number \( h \) of the different temporal dimensions represented in the multi-version XML database, TempPer has the form \( \{ \text{From}_1: \text{To}_1, \ldots, \text{From}_h: \text{To}_h \} \).

The value of the attribute SemAppl is a list \( \{ T_1.C_1, \ldots, T_n.C_j \} \) of identifiers of the classes to which the tuple is applicable. Such identifiers correspond to the classes \( T_i.C_j \) involved in one of the disjuncts \( \text{appl}(n^{MV}) = \bigwedge T_i.C_j \) making up the applicability of the node \( n^{MV} \) expressed in DNF.

In this way, every multi-version node \( n^{MV} \), whose applicability involves multiple disjuncts and whose lifetime is a temporal element made up of a union of periods, is represented through several value-equivalent tuples, with the same values of explicit attributes DocId, LeftPos:RightPos, LevelNum, and the values of implicit attributes SemAppl and TempPer corresponding to one of the different (applicability disjunct, temporal period) pairs deriving from the decomposition of the \( \text{applicability}(n^{MV}) \times \text{lifetime}(n^{MV}) \) pertinence.

![Recommendations](image1.png)

In this framework, before a multi-version XML guideline is actually inserted in the indices, it undergoes a pre-processing phase where the lifetime and applicability of each node is extracted from the timestamps and applicability annotations associated with it. Fig. 10 illustrates the structure of the four indices built for the reference example.
3.5. A Holistic Technology for the Multi-version Slice Operator: Temporal Aspects

The multi-version slice operator can be implemented by applying minimal changes to the holistic twig join architecture. The multi-version XML database is recorded in the multi-version inverted indices which substitute the conventional inverted index at the lower level of the architecture and, thus, the nodes in the stacks are represented by the combination of position, temporal and semantic attributes. Given a personalization query expressed as \texttt{mv-slice(twig, t-window, a-formula)}, standard holistic twig join algorithms can still be used to match twig patterns and, to this purpose, such algorithms provide an optimal management of the stacks. Temporal and semantic consistency, on the contrary, must be enforced on the returned slice with ad-hoc solutions. In this section, we will focus on the temporal aspects involved in the implementation of the slice operator and describe how we adapted the solution proposed in [74]. For clarity of presentation, in this section we will manage tuples as if they were projected on their explicit and temporal attributes only. Hence, any tuple will be referenced here as \((D,L,R,N|T)\). Semantic aspects will be reintroduced and dealt with in the next section.

Temporal consistency could be checked on each structurally consistent answer, that is after the overall query processing exemplified in Fig. 9, at the SOL level. In particular, for each potential slice \(((D,L_i;R_i,N_i|T_i)),..., (D,L_k;R_k,N_k|T_k))\), it is necessary to check that the intersection of the periods represented by the values \(T_i,\ldots,T_k\) overlaps the temporal window of the query (overlap constraint in the following). In this way, we have described a “first step” towards the realization of the temporal aspects of an XML query processor. On the other hand, the performances of this first solution are strictly related to the peculiarities of the underlying database. Indeed, XML documents usually contain huge amounts of nodes and this is absolutely true in the medical context where multi-version guideline documents record the history of the applied changes. Thus, the holistic twig join algorithms can produce a lot of answers which are structurally consistent but which are eventually discarded as they are not temporally consistent with the query. This situation implies the execution of useless computations related to an uncontrolled growth of the number of tuples pushed on the stacks, regardless of their temporal consistency.

Besides this straightforward solution, we devised alternative ones, which act at the different levels of the architecture, aimed at limiting the number of useless nodes the algorithms push into the stacks. The resulting reference architecture is slightly different from the one presented in Fig. 8. Indeed, in our proposal, any timestamped node whose lifetime is a temporal element is usually encoded into several tuples (e.g., see the encoding of node E in the reference example). Thus, at level L1, each node \(n_q\) must be interpreted as “the set of tuples encoding \(n_q\)”: they are stored in buffer \(B_q\) and step 3 of the HTJ algorithms empties \(B_q\) and pushes the tuples in the stack \(S_q\).
However, not all tuples which enter level L1 will in the end belong to the selected set of slices. In particular, some of them are discarded thanks to the following constraint:

**Overlap Constraint.** Let \((D,L:R,N|T)\) be a tuple belonging to the temporal inverted index \(I_q\). Then \((D,L:R,N|T)\) will not belong to any selected slice if the intersection of its temporal pertinence with the temporal window \(t\)-window is empty.

Thus, the overlap constraint can be exploited to prevent the insertion of useless nodes into the stacks by acting at level L1 and L2 of the architecture. At level L2 we act at step 3 of the HTJ algorithms by simply avoiding pushing into the stack \(S_q\) any temporal tuple \((D,L:R,N|T)\) encoding the next node to be processed which satisfies the overlap constraint, that is such that \(T \cap t\)-window = \(\emptyset\). On the other hand, at level L1, we avoid loading into a buffer \(B_q\) any temporal tuple encoding \(n_q\) to which proposition 1 applies. More precisely, given the LeftPos value of the last processed node, say \(CurLeftPos\), we only load tuples \((D,L:R,N|T)\) such that \(L\) is the minimum value greater than \(CurLeftPos\) and \(T\) intersects \(t\)-window. To this purpose, our solution improves the performances of range-interval selection queries on the temporal inverted indices by adopting time-key B'-tree indices which combine the LeftPos attribute with the attributes \(\text{From}_j:To_j\) representing each temporal dimension in the TempPer implicit attribute. In particular, B'-tree indices are built on the \((\text{LeftPos}, \text{To}_j)\) attribute pair, which logically cluster data primarily on the LeftPos attribute and secondarily on the \(\text{To}_j\) one. In this way, we can take advantage of sequential I/O as index leaf pages are linked and records in them are ordered. We start with the first leaf page that contains a LeftPos value greater than \(CurLeftPos\) and a \(\text{To}_j\) value greater than or equal to \(t\)-window|\(\text{From}_j\), which is the projection of the period \(t\)-window on the interval start time \(\text{From}_j\). Then we proceed by loading the records until the leaf page with the next LeftPos value or with a \(\text{From}_j\) value greater than \(t\)-window|\(\text{To}_j\) is met. This has the effect of selecting each tuple \((D,L:R,N|T)\) where \(L\) is the smallest value greater than \(CurLeftPos\) and its period \(T|\text{From}_j:To_j\) intersect the period \(t\)-window|\(\text{From}_j:To_j\), as \(T|\text{To}_j \geq t\)-window|\(\text{From}_j\) and \(T|\text{From}_j \leq t\)-window|\(\text{To}_j\).

Furthermore, besides levels L1 and L2, the overlap constraint also allows us to act between level L0 and level L1 of the architecture. In fact, it is possible to add an intermediate level between L0 and L1, which we call “under L1” (UL1), where the only tuples surviving the overlap constraint are selected from each multi-version inverted index, are ordered on the basis of their (DocId, LeftPos) values and then pushed into the buffers. With a trick similar to that used at level L1, in order to speed up the selection, we exploit B'-tree indices built on one temporal dimension. Notice that this solution manages buffers as streams of
tuples and, thus, it provides interesting efficiency improvements only when the temporal window is quite selective.


As far as the semantic aspects of the \texttt{mv-slice} operator are concerned, we provide some initial solutions to the problem of selecting only slices implying the \texttt{app-constraint} formula in a correct and efficient fashion. Similarly to the temporal case, our ultimate goal is to introduce the changes that are needed to make the holistic twig join technology efficiently deal with semantic aspects. Obviously, the base case is to check applicability constraints at the SOL level. However, this would mean to waste precious CPU time and main memory space for dealing with node patterns which are structurally but not semantically consistent with the query. In an effort to optimize the personalized access to XML guideline documents with limited applicability, the proposed solutions will act at the different levels of the architecture shown in Fig. 8 in order to limit access to semantically useless nodes. All these solutions rely on the idea of arranging the data nodes of the multi-version XML guideline repository and the applicability constraints of the submitted query in the pre-order/post-order planes representing the reference taxonomies. As we have seen, taxonomies obey to the inheritance relation and the pre- and post-order ranks allow us to determine the ancestor-descendant relationship between any pair of applicability classes in constant time.

In this context, each taxonomy $T_i$ corresponds to a pre/post-order plane where document nodes are arranged as follows. Each multi-version node $n^{MV}$ whose semantic annotation \texttt{applicability(n$^{MV}$)} contains a class $T_i.C_j$ is represented in the $T_i$’s plane by the point having the pre- and post-order coordinates of $T_i.C_j$. For instance, the representation of the multi-version XML guideline shown in Fig. 7 in the pre-/post-order planes of the reference taxonomies of Fig. 1 is shown in Fig. 11. Notice that a multi-version node lies in as many planes as many semantic dimensions are considered (i.e., number of taxonomies). At the same time, in each plane, it can be represented by as many points as many classes it is applicable to (see e.g., nodes G and H in the figure). A similar approach is followed to represent the applicability constraints \texttt{app-constraint} passed to the \texttt{mv-slice} operator. In particular, each navigational pattern $[T_j.C_i : \text{depth}_i]$ is represented in the $T_j$’s plane by the rectangle whose lower right corner corresponds to $C_i$ and the upper left corner is the $C_i$’s ancestor reachable with $\text{depth}_i$ steps in $T_j$. Similarly, whenever a navigational pattern $[T_j.C_i : \text{depth}_i]$ is in negated form, \texttt{NOT} $[T_j.C_i : \text{depth}_i]$, it is represented as the plane part outside the corresponding rectangle. For ease of reference, the former is named inclusion rectangle while the latter is
named exclusion rectangle. Fig. 11 also shows the rectangles which correspond to the navigational patterns \([T_1.C_1:1]\) and \([T_2.C_2:1]\) of the query example.

Fig. 11. The pre-/post-order planes of the reference taxonomies \(T_1\) (left) and \(T_2\) (right)

A multi-version node \(n^{MV}\) is semantically consistent if one of the disjuncts of \(\text{applicability}(n^{MV})\) in DNF implies \textbf{a-formula}. Our method consists of efficiently selecting all and only the multi-version nodes that are semantically consistent through their encoding in the pre/post-order space. The basic algorithm is to access each node only once. Specifically, the algorithm executes the following steps:

1. **a-formula** is translated in its equivalent DNF, that is as a disjunction \(\bigvee_i CF_i\) of conjunctive formulas, where \(CF_i = \lnot \mathcal{R}(T_{ij}.C_{ij}:depth_{ij}') \land \lnot \mathcal{R}(T_{ij}.C_{ij}:depth_{ij}) \land \ldots \land \lnot \mathcal{R}(T_{ik}.C_{ik}:depth_{ik}')\) and each of the conjuncts can be in negated form;

2. The regions in each pre/post-plane encoding the applicability constraint **a-formula** are identified and the regions in the same \(CF_i\) (belonging to the same plane or to different planes) are linked to each other;

3. Placement of regions is exploited to smartly access the semantically pertinent nodes only.

From step 1 and owing to (iv) of 3.2, it follows that \(n^{MV}\) is semantically consistent if one of its applicability disjuncts \(\text{appl}(n^{MV})\) implies at least one conjunctive formula, say \(CF_r\). In the pre/post-order space this notion translates into the constraint that the encoding of the node must be contained in all the \(CF_r\)'s inclusion rectangles \([T_{rth}.C_{rth}:depth_{rth}]\) and, at the same time, must lie outside all its exclusion rectangles NOT \([T_{rth}.C_{rth}:depth_{rth}]\).

As far as the second step is concerned, with the term region we precisely mean a non-overlapping portion of the plane. Thus, for each taxonomy \(T_j\) specified in **a-formula**, we first represent the rectangles in the \(T_j\)'s plane, each rectangle corresponding to the intersection of the navigational patterns
$[T, C_i: \text{depth}_i]$ contained in every a-formula $CF_i$ involving $T_j$. All the inclusion and exclusion rectangles involved in the same $CF_i$ are linked together. Then, regions are derived from the identified rectangles. Specifically, we first identify the horizontal bounds of each region induced by the post-order values of the rectangles. For instance, in Fig. 11 this is $[2,4]$ in the $T_1$ plane and $[1,7]$ for the $T_2$ plane. Notice that, for the sake of simplicity, the planes of our reference example contain only one rectangle each. However, in real scenarios, the rectangles in each plane can also overlap and thus the horizontal bound of each region are derived from the post-order boundaries of the involved rectangles. Then, we detect the vertical bounds of each region as the minimum and the maximum pre-order values of the rectangles overlapping the region’s horizontal bounds. For the sample region in Fig. 11 this is $[4,5]$ in the $T_1$ plane and $[1,2]$ for the $T_2$ plane. Finally, all the regions are ordered in increasing order of their post-order ranks, thus resulting in a sequence of non-overlapping regions: $(T_j, [bpost_{R_1}, epost_{R_1}] \times [bpre_{R_1}, epre_{R_1}])$, …, $[bpost_{R_n}, epost_{R_n}] \times [bpre_{R_n}, epre_{R_n}])$ (i.e., the region $[2,4] \times [4,5]$ for the $T_1$ plane and the region $[1,7] \times [1,2]$ for the $T_2$ plane in our example).

In the third step, the resulting regions can be exploited at different levels of the holistic architecture to efficiently retain the semantically pertinent tuples only. For instance, at level L1 we avoid loading in any buffer $B_q$ any tuple encoding a node $n_q$ which is out of the region boundaries. In this case, among the available planes, the algorithm starts checking tuples from the plane whose regions fill up the smaller space. More precisely, B’-tree indices are built which logically cluster first on the LeftPos attribute and then on the post-order of each class in SemAppl. In this way, given the LeftPos value of the last processed node, say $CurLeftPos$, it is possible to efficiently sequentially scan only those tuples having the smallest LeftPos value greater than $CurLeftPos$ and laying in the regions. In particular, the access is performed on $(CurLeftPos, bpost_R)$ first, thus allowing us to load the tuples whose pre-order values of SemAppl are contained in $[bpre_R, epre_R]$. Then, the other classes that may be listed in SemAppl are checked with respect to the regions connected to $[bpre_R, epre_R]$. The same process is recursively applied for each region. Similarly, it is possible to act at level L2 by simply avoiding the insertion of tuples which are out of the regions.

While the solutions proposed in Section 3.5 aim at producing temporally consistent node patterns, the ones sketched here guarantee semantic consistency. Therefore, a naïve approach for producing both temporally and semantically consistent solutions could be to intersect the answers returned by the two processes. However, as the two versioning aspects can be dealt with in an orthogonal way, the techniques proposed to enforce semantic consistency can be freely combined with the ones proposed to enforce
temporal consistency at any level of the holistic architecture, which can be chosen to enhance the overall efficiency of multi-version XML guideline personalization engine.

3.7. Experimental Evaluation

A prototype of the personalization engine, implementing various combinations of the techniques presented in the previous sections, has been developed using Java JDK 1.6 and ad-hoc data structures [24]. In this section, the goal of our evaluation is to assess the efficiency of the proposed personalization engine, since, as we have seen in the previous sections, the first real technological problem of managing multi-version guidelines is that their size and structure makes traditional database technology fail in searching them in a fast and scalable way. To this end, we present a selection of the results we obtained on repositories of synthetic guideline documents (previous tests on the same prototype implementation in the legal domain can be found in [74]). The synthetic guidelines used for experiments have been generated with a structure conforming to the GEM guideline schema\(^4\) presented in [14]. Note that, while we chose GEM as the base schema of the synthetically generated documents, any other XML guideline model could have been chosen, since the key aspect for testing efficiency is not the actual contents of the documents but the multi-version structural properties through which we are able to stress the system.

As far as the multi-version aspects are concerned, we applied our multi-version encoding scheme to the guidelines, using three temporal dimensions, by means of a configurable XML generator [74]. On average, each document contains 50÷60 nodes, with a depth level of 10, and 15÷20 of these nodes own 2 or 3 time-stamped versions. A reference collection has been built consisting of 5,000 guideline documents (120 MB). First, we tested the performance of our query processing technique on this collection, with different “evaluation scenarios” corresponding to the solutions discussed in Section 3.5 and with different slicing settings presenting different temporal window selectivities. Then, we evaluated the scalability of our prototype system by testing it on two larger collections, containing 10,000 and 20,000 documents. The prototype implements all the technologies and solutions discussed in Section 3.5 for the management of the temporal aspects of the slice operator [74], which can be extended with the support of semantic aspects as described in Section 3.6. All the experiments were executed on an Intel Core2 Duo 2.4Ghz OSX workstation, equipped with 2GB RAM and a 160GB SATA disk.

\(^4\) http://gem.med.yale.edu/gemschema/
<table>
<thead>
<tr>
<th>Evaluation scenarios:</th>
<th>Execution Time (ms)</th>
<th>Tuples (%) Buffer</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1323</td>
<td>9.43 %</td>
<td>9.12 %</td>
</tr>
<tr>
<td>L2</td>
<td>1838</td>
<td>23.80 %</td>
<td>9.12 %</td>
</tr>
<tr>
<td>SOL</td>
<td>8882</td>
<td>100.00 %</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>High sel</th>
<th>Med sel</th>
<th>Low sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>722</td>
<td>1323</td>
<td>1968</td>
</tr>
<tr>
<td>L2</td>
<td>1002</td>
<td>1838</td>
<td>2395</td>
</tr>
<tr>
<td>UL1</td>
<td>558</td>
<td>2155</td>
<td>2757</td>
</tr>
<tr>
<td>SOL</td>
<td>8727</td>
<td>8882</td>
<td>8888</td>
</tr>
</tbody>
</table>

Tab. I. Performance evaluation with different evaluation scenarios and slice selectivities.

As shown in the top part of Tab. I, we started by testing the slice operator with a default slicing setting (medium selectivity, i.e., 20% of the tuples stored in the temporal inverted indexes involved by the twig pattern intersect the temporal window). The different scenario names represent the level at which our multi-version slicing techniques is applied; in the baseline scenario SOL, none of the discussed solutions and optimizations are employed. In the table, from the left to the right, columns report the execution time, the percentage of tuples that are put in the buffers and in the stacks with respect to the total number of tuples involved in the query evaluation. The best result is obtained with the computation scenario L1, whose execution is more than 6 times faster than the baseline scenario SOL. Such a result clearly shows that exploiting solutions at a low level of the architecture, such as L1, avoids I/O costs due to fetching unnecessary tuples and their further processing costs at the upper levels. The decrease in read tuples from 100% of SOL to just 9.43% of L1 represents a remarkable result in terms of efficiency gain. In scenario L2, the percentages of tuples in buffers are more than twice those in L1, while the overall execution time is about 1.5 times higher. The lower part of Tab. I shows how our multi-version XML query processor responds to the execution of slicing with different selectivity ratios; to this purpose, in addition to a medium selectivity slice, we also considered slice settings with a low and a high selectivity (31% and 1%, respectively). Also in the low selectivity setting, the L1 scenario remains the most efficient, even if the lower selectivity of the temporal window makes the benefits achievable by the L1 solutions less appreciable. Moreover, as one would expect, we see that the UL1 scenario is inefficient with low-selectivity settings, whereas it becomes the best one with the high-selectivity setting. Such behavior is not qualitatively affected by the management of semantic aspects with the solutions outlined in Sec. 3.6.
Fig. 12 reports the scalability performance of our XML query processor in executing the medium selectivity slicing setting for the collections containing 5,000, 10,000 and 20,000 guidelines. The execution time grows linearly in every scenario (notice the logarithmic scales), and with a proportion of approximately 0.75:1 with respect to the number of documents in the best performing scenario L1. This is an important achievement: non-optimized solutions such as the SOL baseline, beside being significantly slower, do not benefit from larger document sets and scale with a 1:1 proportion. The same is true for typical implementations on top of a DBMS [23], where the post-processing phase necessary to deal with the temporal and semantic aspects is forced to work on one document at a time in main memory. Similar tests, where we measured a similar trend, were also performed in the presence of the other slicing settings, showing, thus, good scalability of the personalization engine in every type of query context. As a final note on memory requirements, our tests showed that our implementation required, for each evaluation scenario, less than 5% of the main memory used by approaches relying on standard (relational or XML) DBMS platforms. Also this performance figure is of primary importance when considering a guideline management system running in a highly concurrent environment like a hospital computer system.

![Figure 12: Scalability results for 5,000, 10,000 and 20,000 guidelines.](image)

4. CONCLUSIONS AND FUTURE WORK DIRECTIONS

In this paper, we introduced novel solutions to represent and to provide personalized access to multi-version clinical guidelines, supporting multiple temporal and semantic versioning coordinates. The proposal involves the definition of a multi-version XML data model and the introduction of data structures and algorithms underpinning a new concept of personalization engine. Efficiency and scalability of the approach has been evaluated by means of experiments conducted on a prototype of the personalization engine.
Future work will consider the improvement of the approach to cope with more advanced application requirements (e.g., relaxing of constraint of tree-like ontologies, capturing more complex query specifications, managing multiple ontology versions) and the completion of the technological infrastructure required to set up the personalization platform with the design and implementation of auxiliary services (e.g., for automatic patient classification with respect to the taxonomies, integration with electronic medical record systems, etc.). In particular, support of more advanced query specifications like, for example, applicability criteria involving temporal relationships and explicit value restrictions (e.g., “applicable to patients having during the passed month an average systolic blood pressure between 120 and 140”) as captured in the ERGO approach [36], are currently beyond the capabilities of our approach. However, since value-based attributes can be discretized to be transformed into categorical attributes and used to organize a concept hierarchy (cf., the age classes in the demographic taxonomy), an automatic reasoner could be used to map advanced value restrictions specified in a query onto classes of the supported taxonomies, in a manner similar to the classification of eligibility criteria proposed in [36]. Future work will explore this possibility.

Further work will also include the assessment of our proposal in a concrete working environment, with a repository of real clinical guidelines and real users evaluating the clinical utility and reference of the proposed guideline multidimensional versioning and personalization methods, which is a fundamental step in developing a tool aimed at improving and optimizing healthcare activities. This final aim could be facilitated by the integration of our developed components into an existing comprehensive framework like that making up the DeGeL approach.

References


