Flavio Rizzolo · Alejandro A. Vaisman

Temporal XML: Modeling, Indexing, and Query Processing

Abstract In this paper we address the problem of modeling and implementing temporal data in XML. We propose a data model for tracking historical information in an XML document and for recovering the state of the document as of any given time. We study the temporal constraints imposed by the data model, and present algorithms for validating a temporal XML document against these constraints, along with methods for fixing inconsistent documents. In addition, we discuss different ways of mapping the abstract representation into a temporal XML document, and introduce TXPath, a temporal XML query language that extends XPath 2.0.

In the second part of the paper, we present our approach for summarizing and indexing temporal XML documents. In particular we show that by indexing *continuous paths,* i.e., paths that are valid continuously during a certain interval in a temporal XML graph, we can dramatically increase query performance. To achieve this, we introduce a new class of summaries, denoted *TSummary,* that adds the time dimension to the well-known path summarization schemes. Within this framework, we present two new summaries: *LCP* and *Interval* summaries. The indexing scheme, denoted TempIndex, integrates these summaries with additional data structures. We give a query processing strategy based on TempIndex and a type of ancestor-descendant encoding, denoted temporal interval encoding. We present a persistent implementation of TempIndex, and a comparison against a system based on a non-temporal path index, and one based on DOM. Finally, we sketch a language for updates, and show that the cost of updating the index is compatible with real-world requirements.

Keywords : XML, Temporal databases, Semistructured data, Structural summaries, XPath.

University of Toronto
40 St. George St. Bahen Center for Information Technology
University of Toronto,
Toronto, Ontario M5S 2E4 Canada
Tel.: +1-416-946-0398
Fax: +123-45-678910
E-mail: flavio@cs.toronto.edu

Universidad de Chile and Universidad de Buenos Aires
Ciudad Universitaria, Pabellon I, Buenos Aires, Argentina
Tel.:+5411-4576-3359
Fax: +5411-4576-3359
E-mail: avaisman@dc.uba.ar

1 Introduction

The topic of representing, querying and updating temporal information has received little attention in the XML literature. Nevertheless, time is present in almost any real-world application, especially in web and e-business applications. In this paper we will show how temporal database concepts [52, 19] can be applied to define, query and manage temporal XML documents, i.e., XML documents that can be navigated across time.

The graph depicted in Figure 1 is an abstract representation of a temporal XML document for a portion of the NBA[1] database. We will be using this example throughout the paper. The league is composed of franchises that maintain teams, and each team has a set of players that may change over time. Some franchises may have players directly associated to them, not included in teams. The database also records some statistics for each player. Note some of the dynamics that this example graph models: players move from one franchise to another, usually from year to year, while their statistics change from match to match. For instance, in this database, node 14 represents player Williams. The dashed line between nodes 2 and 14, labeled [0,22], indicates that he played for the Orlando Magic between instants '0' and '22'. After that, he moved to the Toronto Raptors (a team corresponding to this franchise is represented by node 5), where he is currently playing. This is represented by the solid line joining nodes 5 and 14, labeled

[1] National Basketball Association, a professional basketball league

[23,Now]. Notice that in spite of the change of franchise, there is only one node for each player, which contains all the player's information. Thus, regardless of the franchise he played for, the graph shows that Williams scored twenty-two points throughout his career. As another example, node 24 represents player Garrity, who scored fifteen points between instants '0' and '10', and twelve points since then. In the next sections we will describe in detail the components of Figure 1.

The information contained in the abstract representation of a temporal document presented in Figure 1 allows to traverse the history of the NBA stored using this single document. We can then (a) query the state of the database at a certain point in time (technically, a *snapshot* of the document); or (b) pose temporal queries like "players who played for the Toronto Raptors continuously since at least the year 2000" or "name of the players who were with the Orlando Magic when McGrady joined the franchise for the first time". For these kinds of queries we provide in Section 7 an indexing scheme and in Section 8 efficient query evaluation techniques.

Other approaches (based on versioning) store only the information at some point in time, and use edit scripts and *diff* algorithms to reconstruct the histories of the document. In Section 2 we discuss the different ways of tackling this problem, and show that our approach can do better than versioning, mainly when we expect frequent updates that may cause a query to span over many versions.

In the first part of this work we address the problem of modeling and implementing temporal features in XML documents. We begin by defining an abstract model for temporal XML documents as a graph with annotated edges of two kinds: *containment* edges, describing element nesting and attribute values, and *reference* edges describing IDREF to ID references. Both kinds of edges are annotated with *temporal elements* (actually, for the sake of clarity, we will work with single intervals). Next, we study *consistency* conditions for temporal XML documents based on our data model (although our approach is general enough to be extended to other data models). We present algorithms for checking consistency, study their computational complexity, and discuss solutions for fixing inconsistencies. To the best of our knowledge, this is the first contribution on consistency of temporal XML documents, although this has been studied for non-temporal XML in [20]. In addition, we discuss different ways of mapping the abstract temporal model into a concrete XML document and, finally, introduce TXPath, a query language that extends XPath for supporting temporal queries.

In the second part of the paper we present a framework for *structural summaries* of temporal XML documents, and study an indexing scheme, denoted TempIndex, introduced in previous work [37]. Several structural summaries have been proposed in order to optimize path query evaluation over non-temporal data graphs. Some recent works on structural summaries in the XML context include [25,38,31,33,45]. Most of these proposals keep record of the paths in the XML data by summarizing path information in different ways. They construct a concise representation of the XML nodes based on their labels, usually a labeled graph. Although indexing label paths on temporal documents helps to reduce the search space, our experiments show that computing paths within a given time interval is quite expensive even in the presence of traditional path indexes. One possible solution is to integrate the temporal dimension into the indexing scheme in order to obtain better performance. TempIndex accomplishes this integration by summarizing label paths together with temporal intervals and *continuous paths* (paths that are valid continuously during a certain interval). Finally, we sketch a language for updates in XML, and show how consistency checking affects the definition of update operators.

This paper considerably updates and extends the work presented in [37]. Section 2 has been expanded, providing a better comparison of our model with other proposals. Section 3 gives a more detailed discussion of the data model. Section 4 is completely new, presenting an in-depth study of temporal consistency issues. Section 5 is also new. Sections 7 and 8 have been substantially modified: the notion of structural summary, and the introduction of the *TSummary* class of summaries give a totally new framework for the indexing scheme. Last, but not least, we now present a *persistent* implementation, which allows managing larger documents, and makes the results presented in Section 10 much more relevant and significant.

The remainder of the paper is organized as follows: in Section 2 we review previous efforts in temporal semistructured/XML data and non-temporal structural summaries indexes. In Section 3 we introduce the temporal data model. Section 4 presents an in-depth study of the consistency conditions required by the data model, algorithms for checking these conditions, and methods for fixing (if needed) inconsistent documents. Section 5 presents four alternatives for mapping the abstract representation to a concrete XML document. TXPath is introduced in Section 6. We present the TSummary framework and the TempIndex scheme in Section 7, and Section 8 explains how to use this scheme in query processing. Section 9 discusses updates in temporal XML documents, and update management in TempIndex. Implementation and testing results are presented in Section 10. We conclude in Section 11.

2 Related Work

Temporal Relational Databases. Temporal relational database management has been extensively studied in the literature, including data models [52] and query languages [10] (like TSQL2 [51]). However, most proposals

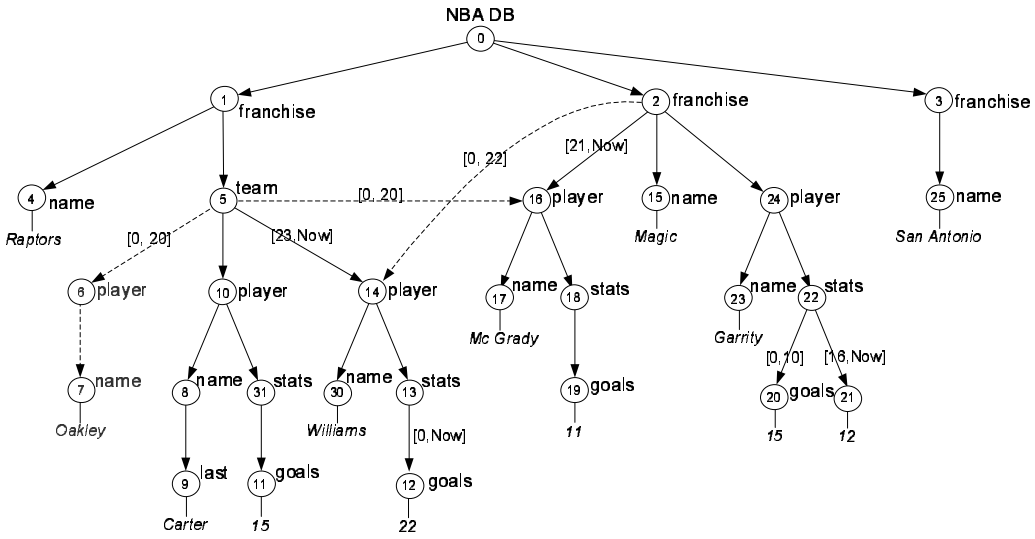

Fig. 1 Temporal XML document for a portion of the NBA database

in the relational database framework require complex extensions to SQL, and commercial databases provide only limited built-in support for temporal information management.

Temporal Semistructured Databases. A model for managing historical semistructured data was proposed by Chawathe *et al.* [6]. They extend the Object Exchange Model (OEM) [7] with the ability to represent updates and to keep track of them by means of "deltas". However, they do not apply this work to XML. Along the same lines, Oliboni *et al.* [40] proposed a graphical data model and query language for semistructured data supporting transaction time, by means of attaching an interval of validity to the objects of the model. Dyreson *et al.* [18] went further, allowing annotations on the edges of the database graph that can refer not only to valid or transaction times, but other kinds of metadata as well.

Temporal XML. In the last few years, many proposals have addressed the problem of maintaining versions of XML documents. Grandi [26] provides a good index to bibliography on temporal aspects in the Web. Updates to XML in a non-temporal framework has been first studied by Tatarinov *et al.* [53]. They proposed a language for updating XML documents as an extension to XQuery [59]. A model for granting access to temporal XML documents was introduced by De Capitani [14]; however, the focus here is the authorization model, not the temporal features of the document. Grandi and Mandreoli [27] presented an infrastructure for managing temporal web documents. Amagasa *et al.* [2] introduced a temporal data model based on XPath, but not a model for updates, nor a query language taking advantage of the temporal model. Dyreson [17] proposed an extension to XPath with support for transaction time by means of the addition of several temporal axes for specifying temporal directions. Their focus is on document versioning over the web in the absence of explicit timestamps. Manukyan *et al.* [35] attempted formalizing temporal constituents of XML documents. They do not address querying temporal XML documents, neither discuss implementation issues. Chien *et al.* [8,9] proposed update and versioning schemes for XML. First, they presented an edit-based schema [8] in which the most current version of the document is maintained, and reverse edit scripts that allow moving backward in version time. They later moved to a scheme where version management is performed by keeping references to the maximal unchanged subtree in the previous version [9], sharing unchanged elements among versions. The main difference between their approach and ours is that we maintain a single temporal document from which versions can be extracted when needed. We believe this is better for scenarios where changes are frequent and only affect a few elements of the document. In this situation, creating a new physical version each time an update occurs may lead to large overheads when processing temporal queries that span multiple versions. A similar approach was followed by Marian *et al.* [36]. Their goal was detecting, managing and notifying changes in web data warehouses of XML data in the context of the *Xyleme* project [1], a project aimed at building a dynamic World Wide Web data warehouse. The idea here is that Xyleme periodically refreshes its data and computes the changes using a *diff* algorithm. All nodes are assigned a Xyleme ID, which is independent of the ID attributes that the document may contain. It follows that Xyleme's goals and requirements differ from ours. Wang and Zaniolo have also proposed solutions for the Web Warehousing problem [55,56]. In [55] they proposed a valid time model that represents successive versions of a document as an XML document (implementing a temporally grouped data model) which is then queried using

XQuery or any other XML query language. The latter is the strongest point of this approach. They provide versioning using the special attributes `vstart` and `vend` (and, in a subsequent paper [56] `tstart` and `tend`, for handling also transaction time). Their work shows some examples of the kinds of queries that could be addressed with this approach, but there is neither an in-depth study of the model, nor experimental results supporting their claims. Wang *et al.* [57,58] used a similar concept for managing and querying historical databases. They take advantage of the fact that a temporally grouped data model fits well into the XML data model. Thus, they map historical databases to so-called H-documents. This approach allows posing queries in XQuery and evaluate them using a relational database. They provide preliminary experimental results over a very simple example. Although this work overrides the problem of having multiple versions of the same document, and allows querying with any standard XML query language, it is not clear how general this solution is (i.e. how it can be efficiently applied to more involved situations), given the limited semantics of the data model. It seems that the temporal grouping assumption may limit the model to handle particular cases, where no relationships between complex objects change (like in our NBA example). Nothing is said about query optimization using index structures appropriate for temporal information, and updates are only vaguely discussed in [56].

Gergatsoulis and Stavrakas [24] introduced a model for representing changes using an extension to XML denoted MXML (Multidimensional XML), where dimensions are applied to elements and attributes. Queries are not addressed in this work, but the authors claim that queries can be posed after a reduction from MXML to XML.

Our proposal has similarities with the work of Buneman *et al.* [5]. In this work, the authors study data structures specifically suited for keeping historical information about scientific data. They provide a versioning scheme allowing storing all the information in a single document (i.e., the authors also acknowledge the need for avoiding edit scripts when changes are frequent). Seminal in many senses, we think the proposal is limited to relevant although very specific situations and data formats. The authors also present *timestamp trees*, an efficient indexing scheme that allows obtaining a version of the document at some point in time. The proposal supports documents where the changes consist in addition of information, and is not oriented to documents where the relationships between objects change, or when updates are of a kind other than the insertion of elements. Moreover, the scheme requires that each node in the graph representation of the temporal document must be uniquely identified by the path in which it occurs and the values of its subelements (following the concept of XML *keys* discussed in [4]). The authors conclude that, if the document does not have a key system, the proposal requires a *diff* algorithm, turning it into a conventional Source Control Code System (SCCS). Also, the work is oriented to queries asking for document snapshots or histories of elements, which are only a portion of the queries a temporal database must support. The indexing scheme is also oriented to these kinds of queries. Finally, the issue of handling document order is not considered in the paper. We believe the model proposed in the present paper, although having some features in common with the work of Buneman *et al.*, considerably extends and improves their work, overriding its many constraints.

Also close to our ideas, Gao *et al.* [22,23] introduced τXQuery, an extension to XQuery supporting valid time while maintaining the data model unchanged. Queries are translated into XQuery and evaluated by an XQuery engine. Even for simple temporal queries, this approach results in long XQuery programs. Moreover, translating a temporal query into a non-temporal one makes it more difficult to apply query optimization and indexing techniques particularly suited for temporal XML documents. We would like to make it clear here that we do not compare the expressiveness of τXQuery against TXPath (the language we propose), we only point out the different approach. TXPath is not aimed at being a working temporal query language, but a tool for giving insight into the problems that appear when querying temporal XML data that is stored using different models.

It is worth noticing that none of the approaches commented above provides an in-depth study of the problems of working with inconsistent temporal XML documents. Moreover, most of these proposals only define vague consistency conditions for the data models that support them. This is a subject overlooked so far in temporal XML, although in the last few years the topic has been addressed in the non-temporal XML framework (see for example [20]). An important contribution of our work is the study of different ways of tackling consistency in temporal XML documents.

Structural Summaries for XML. Structural summaries for XML data have been proposed in recent years in order to optimize path query evaluation. Most of these proposals keep record of the paths in the XML data by summarizing path information in different ways. They construct a concise representation of the XML nodes based on their labels, usually a labeled graph. Examples of those are region inclusion graphs (RIGs) [13], representative objects (ROs)[39], dataguides [25], reversed dataguides [34], 1-index, 2-index and T-index [38], and more recently, ToXin [46], A(k)-index [33], F&B-Index and F+B-Index [31], and HOPI [49]. Dataguides and ROs group nodes into sets according to the label paths incoming to them (each node may appear more than once in the dataguide if the document instance is not just a tree). RIGs, 1-index, T-index, ToXin, F&B-Index, and F+B-Index, on the other hand, partition the data nodes into equivalence classes (called *extents* in the literature)

so that each node appears only once in the summary. The partition is computed in different ways: according to the node labels (RIGs), the label paths incoming to the nodes (1-index, ToXin, A(k)-index), the label paths going out from the nodes (reversed dataguides), or all of the above (F&B-Index and F+B-Index). The length of the paths in the summary also varies: ToXin, 1-index and F&B-Index summarize paths of any length, whereas A(k)-index and F+B-Index are synopsis of paths of a fixed length. HOPI is the only proposal designed specifically for graph data instances: it materializes the 2-hop cover of the graph.

Other summaries are augmented with *statistical information* of the instance for selectivity estimation, including path/branching distribution (XSKETCH [41,42], fXSKETCH [15]) and value distribution (XCLUSTER [43]). Another proposal contains statistical information for approximate query processing (TREESKETCH [44]).

A few *adaptive* summaries like APEX [11], D(k)-index [45], and M(k)-index [29] use dynamic query workloads to determine the subset of incoming paths to be summarized. APEX is a synopsis of frequently used paths of any length. D(k)-index and M(k)-index, in contrast, summarize variable-length paths based on both the workload and local similarity (the length of each path depends on its location in the XML instance). In addition, updates to structural indexes have been studied in [32] and [61]. It is important to note that although using a non-temporal summary reduces the search space for TXPath queries it does not help with the temporal semantics of the query evaluation.

In previous work [37] we addressed the problem of indexing temporal XML documents and introduced *TempIndex*, an indexing scheme for *continuous paths* that improves temporal query performance. In the second part of this paper we discuss TempIndex in detail.

3 Temporal XML Data Model

First we define a (fairly standard) graph model of an XML document, and then we extend it to a temporal model.

3.1 XML Documents

For our purposes, an XML document is a directed labeled graph. We distinguish several classes of nodes:

- A distinguished node r, the *root* of the document, such that r has no incoming edges, and every node in the graph is reachable from r.
- *Value nodes*: nodes representing values (text or numeric). They have no outgoing edges, and have exactly one incoming edge, from attribute or element nodes (or from the root).
- *Attribute nodes*: labeled with the name of an attribute, plus possibly one 'ID' or 'REF' annotation.
- *Element nodes:* labeled with an element tag, and containing outgoing links to attribute nodes, value nodes, and other element nodes.

Each node is uniquely identified by an integer, the *node number,* and is described by a string, the *node label.* Edges in the document graph are constrained to be either *containment edges* or *reference edges.* A containment edge $e_c(n_i, n_j)$ joins two nodes n_i and n_j such that: (a) n_i is either r or an element node, and n_j is an attribute node, a value node or another element node; or (b) n_i is an attribute node, and n_j is a value node containing the value for the attribute. Attribute nodes must have exactly one outgoing containment edge (to the attribute's value). A reference edge $e_r(n_i, n_j)$ links an attribute node n_i of type REF, with an element node n_j. Finally, node and edge types in our model allow mixed content, i.e. an element node may have different kinds of child nodes, including more than one value node.

3.2 Temporal XML Documents

The mechanism we use for adding the time dimension to document graphs consists in labeling edges with intervals. We consider time as a discrete, linearly ordered domain. An ordered pair $[a, b]$ of time points, with $a \leq b$, denotes the closed interval from a to b. A set of such intervals is called a *temporal element.* In what follows we will only consider that edges are labeled with single intervals instead of temporal elements. Later in the paper we will justify this decision. As is common in temporal databases, the current time point will be represented with the distinguished word '*Now*'. The document creation instant will be denoted t_0.

3.2.1 Time Labels

We extend the document graph model with temporal labels. A *temporal label* is an interval T_{e_c} labeling a containment edge e_c or reference edge e_r, respectively. The meaning of this label is that given an edge e_c between nodes n_i and n_j, T_{e_c} will represent the time period where the element represented by n_j was contained in the element represented by n_i. In this paper we will work with the *transaction time* of the containment relation. Although we do not deal with *valid time*, it could be addressed in an analogous way. Moreover, we will show that a slight modification to the updates we propose would suffice for supporting a limited notion of valid time. For a reference edge e_r, T_{e_r} represents the *transaction* time of the reference. Edges labeled with temporal labels will be called *temporal edges.* In general, if an edge e is labeled with a temporal label T_e, we will use $T_e.TO$ and $T_e.FROM$ to refer to the endpoints of the interval T_e.

We say that two temporal labels T_{e_i} and T_{e_j} are *consecutive* if $T_{e_j}.FROM{=}T_{e_i}.TO+1$. Note that working with single intervals instead of temporal elements (i.e., sets of intervals) imposes some constraints to the model, which are discussed in Section 3.3.

Definition 1 (Current Nodes and Edges) A temporal containment (reference) edge such that $T_e.TO = Now$ is called a *current* containment (reference) edge. A node is called *current* if one of its incoming containment edges is current. (As we will see below, at most one incoming containment edge can be current.)

3.2.2 Attribute Nodes

In the XML data model, attributes must be unique. This limitation influences the way a temporal data model supports these kinds of nodes. We may (a) disallow attributes to vary over time; (b) treat them as elements of a special kind. We chose the second option. From a formal modeling point of view, we make no difference between an attribute and an element node (except that attribute nodes cannot contain other elements). From a practical point of view, we will define a special element, denoted `<ATTRIBUTE>`. Consider, for example, an element `<person>` representing a woman, with an attribute called *last name*; the value for this attribute will change if she marries. This will be treated as follows. (We will explain the syntax later in the paper). At instant t_0 the element `<person>` looks like:

```
<person name="Maria">
  <ATTRIBUTES>
    <last name Time:From="0" Time:To="Now">
      Perez
    </last name>
  </ATTRIBUTES>
</person> ...
```

After marrying at time t_1, the element will contain:

```
<person name="Maria">
  <ATTRIBUTES>
    <lastname Time:From="0" Time:To="t1-1">
      Perez
    </last name>
    <last name Time:From="t1" Time:To="Now">
      Perez-Gomez
    </last name>
  </ATTRIBUTES>
</person> ...
```

3.2.3 Temporal Data Model for XML

We are now ready to formally define a temporal XML document. First, we introduce the notion of *lifespan* of a node.

Definition 2 (Lifespan of a Node) The *lifespan* of a node n, denoted $lifespan(n)$, is the union of the temporal elements of all the containment edges incoming to the node. The lifespan of the root is the interval $[t_0, Now]$.

Example 1 Consider our running example, the NBA database of Figure 1. The fact that McGrady played for the Orlando Magic between instant '21' and the current time, is represented by the current containment edge $(2, 16)$. The lifespan of node 16 is the union of the elements [0,20] (the temporal label of the incoming containment edge between nodes 5 and 16) and [21,Now] (the label of the *current* incoming containment edge). To simplify the figures, we omit all temporal labels of the form $[t_0, Now]$.

The definitions above, imply some consistency conditions that a graph must satisfy in order to be a temporal XML document. The following definition spells these conditions out.

Definition 3 (Temporal XML Document) A *Temporal XML Document* is a document graph augmented with temporal labels, that satisfies the following conditions:

1. The union of the temporal labels of the containment edges outgoing from a node is contained in the lifespan of the node.
2. The temporal labels of the containment edges incoming to a node are consecutive.
3. For any time instant t, the sub-graph composed by all containment edges e_c such that $t \in \mathrm{T}_{e_c}$ is a tree with root r. We call this subgraph a *snapshot* of the document at time t, denoted $\mathcal{D}(t)$.
4. For any containment edge $e_c(n_i, n_j, T_{e_c})$, if n_j is a node of type ID, the time label of e_c is the same as the lifespan of n_i; moreover, if there are two elements in the document with the same value for an ID attribute, both elements are the same. In other words, the ID of a node remains constant for all the snapshots of the document.
5. For any containment edge $e_c(n_i, n_j, T_{e_c})$, if n_j is an attribute of type REF, such that there exists a reference edge $e_r(n_j, n_k, T_{e_r})$, then $T_{e_c} = T_{e_r}$ holds.
6. Let $e_r(n_i, n_j, T_{e_r})$ be a reference edge. Then, $\mathrm{T}_{e_r} \subseteq lifespan(n_j)$ holds.

3.3 Discussion

We will discuss some characteristics of the model, and some assumptions we have made.

The second condition in Definition 3 implies that we will be working with plain intervals instead of temporal elements (i.e. sets of intervals). This assumption simplifies the presentation and makes the implementations more efficient. Our definitions and theorems can be, however, extended to the case of temporal elements. There are, of course, semantic and practical consequences of our decision. For example, suppose we want to represent the fact that Michael Jordan played for the Chicago Bulls between 1996 and 1998 (i.e., there is a node for the Bulls,

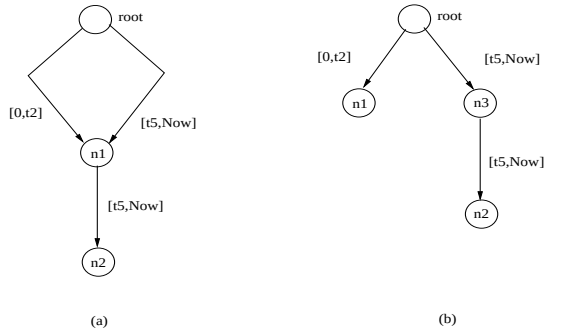

Fig. 2 (a) A gap in lifespan of node n1; (b) A possible solution

another one for Jordan, and an edge between them, labeled with the interval [1996,1998]); then he retired, and after a year he resumed his career. As the model requires that the edges incoming to a node must be consecutive, we cannot represent this situation adding an edge labeled [2000, Now]. A solution could be to create a parent node for 'retired' players, with an edge to the Jordan's node (labeled [1999,1999]), and then, again an edge from the Chicago Bulls' node to the Jordan node, labeled [2000,Now]. We can see that this solution does not generate a significant problem (we may even think, in this case, that it could be a natural way of representing the situation). Another solution, more syntactically oriented (and more likely to be used if the non-consecutiveness came from an inconsistency in the document), can be to duplicate the node with temporal gaps in the labels of its incoming edges. An abstract example is shown in Figure 2.

Remark 1 There is no condition preventing more than one edge between the same two nodes. If they are consecutive, we assume they are coalesced into a single node.

Note that the first constraint in Definition 3 implies that, even though containment edges can only represent containment relations of the same kind in a particular instant, this containment relationship can be a different one in another instant. For example, in the NBA document, a node for McGrady has incoming containment edges from the "team" and "franchise" elements. For any other relationship occurring at the same time we need to use reference edges.

The constraint of having a unique ID throughout the whole history of the document allows overriding many of the restrictions present in [5]. However, it introduces other kinds of problems. Let us suppose that in our running example we would like to represent the same information in a different way, namely with the franchises elements "below" the player nodes (e.g., below the McGrady node we find the Raptors and Magic nodes, with the corresponding temporal labels over the containment edges). The problem here is that the ID attribute would not identify a franchise node (two instances of the same franchise, in the same snapshot, will have different IDs). In this case, even though temporal queries could be answered, we are losing the desirable property of having all the information for a franchise in the same node. Here, the temporal key for a franchise should be the value node containing its name. In the remainder of the paper we will assume that all documents comply with the constraint that, in each snapshot, containment relationships are many-to-one from child to parent nodes (like in Figure 1). In other words, all nodes in a path of containment edges are relative keys (in a snapshot) in the sense of [4]. This, along with the ID constraint, allows identifying a node throughout the document's history.

Definition 4 (Current Subtree) We denote $\mathcal{D}_c$ the subgraph of the temporal XML document $\mathcal{D}$ containing no reference edges. Given a temporal XML document $\mathcal{D}$, and a current node n, the *current subtree of* n, is the subtree of $\mathcal{D}_c(Now)$ with root n.

In the remainder of the paper, for the sake of simplicity, we will consider only *containment* edges, although all the concepts can be extended to consider also *reference* edges.

Definition 5 (Continuous Path and Maximal Continuous Path) A *continuous path* (cp) with interval T from node n_1 to node n_k in a temporal document graph is a sequence $(n_1, \ldots, n_k, T)$ of k nodes and an interval T such that there is a sequence of containment edges of the form $e_1(n_1, n_2, T_1)$, $e_2(n_2, n_3, T_2)$, $\ldots$, $e_k(n_{k-1}, n_k, T_k)$, such that $T = \bigcap_{i=1,k} T_i$. We say there is a *maximal continuous path* (mcp) with interval T from node n_1 to node n_k if T is the union of a maximal set of consecutive intervals T_i such that there is a continuous path from n_1 to n_k with interval T_i.

Example 2 Consider Figure 3. There is only one mcp from node $team(t1)$ to $goals(g3)$, with interval $[99, 02]$. There are 2 mcp's from node $team(t1)$ to $player(p1)$, with intervals $[01, Now]$ and $[95, 97]$. There are 3 continuous paths from the root to $player(p1)$, with intervals $[95, 97]$, $[98, 00]$, and $[01, Now]$; since these are consecutive, they produce a single mcp with interval $[95, Now]$.

An interesting property of mcp's is that they can be computed visiting each node only once. We will take advantage of this property for query processing (see Section 8). Let us consider two nodes n_1, n_k. Let $\mathcal{N}$ be the set of nodes $n_{i,i\neq 1,i\neq k}$ such that there is a continuous path from n_1 to n_i, with interval T_{n_i}, and there is a containment edge from n_i to n_k, with label T_{e_i}. Thus, each continuous path from n_1 to n_k will have interval $T_i = T_{n_i} \cap T_{e_i}$. The union of the intervals of these continuous paths will be the interval of the mcp between n_1 and n_k, if the intervals are consecutive. This means that all mcp's in a graph can be computed visiting each node only once, starting from the root. For example, in Figure 3, if we know the interval of the mcp between f_1 and p_1 we can

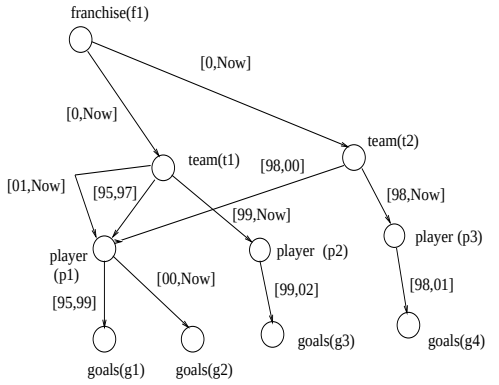

Fig. 3 Maximal Continuous Path

compute the mcp from f_1 to g_1, without visiting the ancestors of p_1. In what follows, except when noted, we will assume that all mcp's are computed from the root.

Document order

In a non-temporal XML document there is a total order between the nodes. A temporal document does not necessarily impose a total order among its nodes, but for any instant t there must be a total order, denoted $<_t$, among the nodes of each snapshot $D(t)$ of document D at time t. In general, for any pair of nodes n_1 and n_2, we may have $n_1 <_{t_1} n_2$, and $n_2 <_{t_2} n_1$, in two different instants t_1 and t_2. However, we can show that there is an interval during which the relative order between n_1 and n_2 does not change. If T_1 is the interval on a continuous path from the root to n_1, and similarly T_2 for n_2, then the ordering between n_1 and n_2 is the same for any instant t in the interval $T_1 \cap T_2$. This is formalized in the following proposition.

Proposition 1 *Let D be a temporal XML document; n_1 and n_2 two nodes in D; $p_1 = (r, \ldots, n_1, T_1)$ and $p_2(r, \ldots, n_2, T_2)$ two continuous paths to n_1 and n_2 with intervals T_1 and T_2, respectively; then, either $n_1 <_t n_2$ for every $t \in T_1 \cap T_2$, or $n_2 <_t n_1$ in every such t.*

Proof By definition of cp (Definition 5) and the third condition of temporal XML document (Definition 3), we know that p_1 is the only path of containment edges to n_1 during interval T_1. (If there were another path of containment edges p_1' to n_1 during any instant t in T_1, then the subgraph composed by all containment edges would not be a tree at instant t.) The same argument can be made about p_2 and n_2 during interval T_2. In particular, p_1 and p_2 are the only paths of containment edges reaching n_1 and n_2 respectively during $T_1 \cap T_2$. Thus, during the entire interval $T_1 \cap T_2$, either $n_1 <_t n_2$ or $n_2 <_t n_1$.

4 Consistency of Temporal XML Documents

Temporal XML documents, as defined in Section 3.2, are subject to continuous updates, which will be studied later in the paper. Such updates must take as input (and return) a consistent XML document. More often than not we will need to check if a temporal document is consistent or not, instead of working with documents built from scratch using update operations. Thus, a study of the cost of such operation is required together with efficient algorithms (not only for checking, but for fixing inconsistencies as well). We will first give consistency conditions for temporal XML documents based on the model presented in the previous section; then, we will propose algorithms for verifying them and give their complexity. In Section 9 we will see how this concepts interplay with the update operators that modify a temporal XML document. Definition 6 below, states the possible inconsistencies in a Temporal XML document.

Definition 6 (Inconsistencies in a Temporal XML Document) The following are the inconsistencies that may violate the conditions stated in Definition 3.

i. There is an outgoing containment edge whose temporal label is outside the node's lifespan.
ii. The temporal labels of the containment edges incoming to a node are not consecutive. Here, the inconsistency may be due to (a) a gap in the temporal labels of some incoming edges; or (b) an overlapping of the temporal labels of some incoming edges.
iii. There is a cycle in some document's snapshot.
iv. There exist more than one node with the same value for the ID attribute.

In what follows we will refer to these types of inconsistencies as inconsistencies of type *i*, type *ii*, and so on. We will not study ID attributes (we will limit ourselves to temporal issues here). Thus, inconsistencies of type *iv* will not be addressed.

Definition 7 (Interval of Inconsistency) Let I be one of the inconsistencies of Definition 6, the *Interval of Inconsistency* of I, denoted I_I, is the closed interval where the inconsistency occurs. The notion of interval of inconsistency is local to I, meaning that there are as many I_I's in a document as inconsistencies occur in it.

Example 3 Figures 4 (a) to (c) show examples of intervals of inconsistency for types *i* to *iii*, respectively. In Figure 4(a) $I_I = [T_4, Now]$ (the temporal label of edge outgoing from n_1 lies outside the lifespan of the node); in Figure 4 (b) $I_I = [T_2, T_4]$ (a gap in node n_2); in Figure 4 (c) there is a cycle in every snapshot within the interval $I_I = [T_4, T_6]$.

Given that computing I_I is, typically, an expensive operation, we have decided to divide the process in two

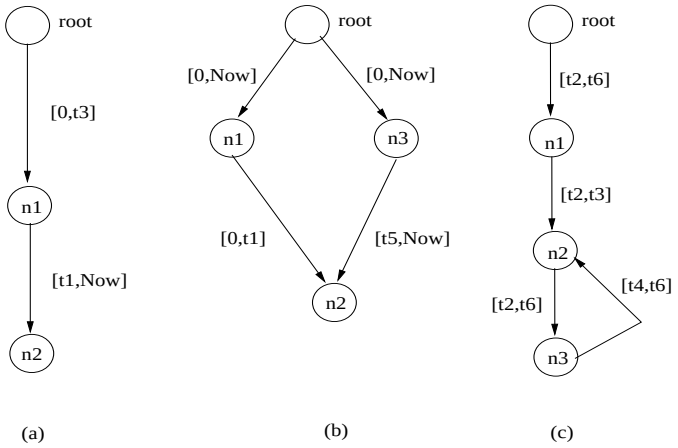

Fig. 4 (a) Inconsistency of type i; (b) Inconsistency of type ii; (c) Inconsistency of type iii

parts: (a) check if the document presents an inconsistency; (b) fix the inconsistency (for this, computing the inconsistency interval is necessary). The user will decide if she wants to execute part (b).

4.1 Checking Consistency

In this section we will study the complexity of checking consistency in Temporal XML documents, and give an algorithm for such task.

Throughout this section we will use the following notion of order: given two intervals T_1 and T_2, if $T_1.TO > T_2.TO$ we will say that T_1 *succeeds* T_2, denoted $T_1 \succ T_2$. Analogously, if $T_1.TO < T_2.TO$, we say that T_1 *precedes* T_2, denoted $T_1 \prec T_2$.

Our algorithm for checking consistency will use the following proposition.

Proposition 2 *Let D be a Temporal XML document where every node has at most one incoming containment edge in every time instant t; if there is a cycle in some interval I_I in D, then, there exists a node n_i such that $T_{mcp}(n_i) \neq lifespan(n_i)$, where $T_{mcp}(n_i)$ is the temporal interval of the mcp between the root and node n_i.*

Proof Assume that there is a cycle in document D during an interval I_I. Let n_i be a node belonging to such cycle. Thus, by definition, we know that $lifespan(n_i) \supset I_I$; however, $T_{mcp}(n_i) \cap I_I = \phi$; if this were not the case, there would be some t such that a path between the root and the node exists, and there cannot exist a node with more than one parent at any instant t.

We will use this property to check consistency condition *iii*. If the property does not hold (assuming that there are no inconsistencies of other types), then, there is a cycle in the document. Algorithm 1 computes the lifespan of a node.

Algorithm 1 (Computing the Lifespan of a Node)

INPUT: A node n
OUTPUT: $I = [FROM, TO]$; Time interval of the lifespan of the node, or null if I cannot be computed.

```
TimeInterval lifespan(node n){
(1) Initialize a list of temporal labels L to null;
(2) I = null;
(3) for each edge e incident to n with label T_e do
(4)     Append (T_e) to L;
(5) Sort L; //using the order relation defined above
(6) I = L[1];
(7) for each i between 1 and length(L) − 1 do
(8)     if L[i].TO + 1 ≠ L[i + 1].FROM then
(9)         Return null;
(10)    I = I ∪ L[i + 1];
(11) end for;
(12) Return I }
```

It can be shown that the lifespan of a node can be computed with complexity

$$O(2deg_{in}(n) + deg_{in}(n) * log(deg_{in}(n)))$$
$$= O(deg_{in}(n)(log(deg_{in}(n)) + 2))$$
$$\approx O(deg_{in}(n) * log(deg_{in}(n)))$$

where $deg_{in}(n)$ is the number of edges incident to n. In the worst case (this considers the case in which all edges are incident to the node), the order of the algorithm is $O(|E| * log(|E|))$; in the average case (all nodes have the same number of incoming edges, i.e. $\frac{|E|}{|V|}$), this reduces to $O(\frac{|E|}{|V|} * log(\frac{|E|}{|V|}))$. In the best case (when each node has only one incoming edge) the lifespan is computed in constant time.

The following algorithm checks inconsistencies of types i and ii.

Algorithm 2 (Checks Inconsistencies of Types i and ii)

INPUT: A temporal XML document D
OUTPUT: True *if D has no inconsistencies of types i and ii;* False *otherwise.*

```
boolean checkNodeConsistency(document D) {
(1) for each node n in D do
(2)     I = lifespan(n);
(3)     if is null(I) and n is not the root then
(4)         Return False;
(5)     for each edge e outgoing from n do
(6)         if T_e is not in I then
(7)             Return False;
(8)     end for;
(9) end for;
(10)Return True;}
```

Lines 5 to 7 check inconsistencies of type i, line 3 checks the occurrence of inconsistencies of type ii (if the intervals of the incoming edges are not consecutive Algorithm 1 returns *null*).

We can see that the main loop iterates at most $|V|$ times (the number of nodes in the document). Lines 2

and 3 check inconsistencies of type i, computing the lifespan of n, with order $O(deg_{in}(n) * log(deg_{in}(n)))$, as explained above. In total, $\sum_{n \in |V|} deg_{in}(n) * log(deg_{in}(n))$. For the average case, $deg_{in}(n)$ equals $\frac{|E|}{|V|}$, yielding: $|E| * log(\frac{|E|}{|V|})$. Lines 5 to 7 compose a loop that repeats for each edge outgoing from a visited node, performing operations of constant order. This, for the average case, results in complexity $O(|V| * \frac{|E|}{|V|})$. The algorithm's order is then: $O(E * log(\frac{|E|}{|V|}) + 1)$.

The next algorithm checks for cycles in a temporal labeled graph.

Algorithm 3 (Finds Cycles in a Document)

INPUT: A temporal XML document D, such that checkNodeConsistency(D) = true
OUTPUT: True *if D has no cycles, otherwise* False

```
boolean checkCycles(D){
(1) Queue nodes = getRoot(D) (a queue,
    initialized with the root of D)
(2) Queue nodesWait = [ ] (empty queue of nodes)
(3) Set traversed(e), usable(e), visited(n) and
    ended(n) to False, for all edges e and nodes
    n in D.
(4) while !empty(nodes) do
(5)     n = first(nodes)
(6)     if !ended(n) then
(7)       labelList = [T_e where e is an edge
          incoming to n and !traversed(e)]
(8)       for each e outgoing from n do
(9)         if !traversed(e) then
(10)          if T_e ∩ T_e' ≠ φ for some e' in labelList then
(11)            usable(e) = False
(12)          else
(13)            usable(e) = True
(14)          end if;
(15)        end for;
(16)    end if;
(17)    for each edge e(n, n_f) and !traversed(e) do
(18)      if (usable(e)||ended(n)) then
(19)        traversed(e) = True
(20)        if traversed(e) = True, ∀e incoming to n_f
            then
(21)          Append n_f to nodes
(22)          ended(n_f) = True
(23)        else
(24)          if !visited(n_f) then
(25)            Append n_f to nodesWait
(26)            visited(n_f) = True
(27)          end if;
(28)        end if;
(29)      end if;
(30)    end for;
(31)    if empty(nodes) and !empty(nodesWait) then
(32)      n = First(nodesWait)
(33)      Append n to nodes
(34)      visited(n) = False
(35)    end if;
(36) end while;
(37) for each node n in D do
(38)      if !ended(n)
(39)        Return False;
(40) end for
(41) Return True }
```

In Algorithm 3, functions $traversed(e)$ and $ended(n)$ apply to edges and nodes, respectively. A node is *ended* when all its incoming edges have been *traversed*. The intuition behind this notion is that, since we are treating isolated inconsistencies, all the edges outgoing from an ended node are *usable* (i.e., can be traversed). There are two queues: (a) *nodes*, which holds all nodes such that all of their incoming edges have already been traversed, and (b) *nodesWait* holding the nodes that have been *visited* but have incoming edges not yet traversed (the *visited* function is used to indicate this). Note that if the document is a tree, the latter queue will always be empty. If the two queues are empty, and all nodes are ended, the document contains no cycle. Conversely, if unended nodes remain, there is a cycle in the document.

Example 4 Let us suppose a graph with nodes *root*, n_1 and n_2. The edges are $e_1(root, n_1, [T_1, T_2])$, $e_2(n_1, n_2, [T_2, T_4])$, and $e_3(n_2, n_1, [T3, T4])$. Clearly, there is a cycle in $[T_3, T_4]$ between n_1 and n_2. When the algorithm reaches n_1 in $[T_1, T_2]$, the node is stored in *nodesWait* since n_1 is not ended. Then, as the queue *nodes* becomes empty, n_1 is removed from *nodesWait* and added to *nodes*. Also, $visited(n_1)$ is set to *False*. However, when the usability of the edge is checked in line (10) of the algorithm, there is a non-empty intersection between the temporal labels, and the edge e_3 is not "usable". Thus, there are no more edges to traverse, and remaining unended nodes exist. Then, the document must contain a cycle.

Theorem 1 *Algorithm 3 finds all temporal cycles in the graph, does not loop indefinitely, and only returns* False *if it finds a cycle.*

Proof 1. *The algorithm has a finite number of loops.* In each main loop the algorithm visits only edges (not yet traversed) outgoing from a node, and adds to a list the nodes such edges are incident to. When all possible edges have been traversed, no node will be added, and the algorithm will stop.

2. a) *The algorithm finds all cycles in the graph.* Let us suppose there a cycle in the graph and it is not detected by the algorithm (i.e., $True$ is returned). Let $n_1 \ldots n_k$ be the nodes in the cycle and T the cycle's interval. Since the algorithm returned $True$, all nodes were "ended", including $n_1 \ldots n_k$, meaning that these nodes were visited in T (because, in order to be ended, all incoming edges must be traversed in the whole interval). Let n_1 be the first node visited in T; it should have been reached from a node n whose incoming edges in T were already traversed. As n_1 was the first one to be visited in such interval, it follows that n does not belong to the cycle; thus, n_1 has a parent outside T and another one inside it, which is a contradiction, because of the precondition stating that no inconsistencies of other kind pre-existed in the document.

b) *The algorithm returns False only if there is a cycle.* Let us suppose there is at least one node n_1 such that $ended(n_1) = False$ and no cycle was found. As there cannot be inconsistencies of type i, all edges incoming to n_1 have a temporal label inside the lifespan of the starting node. Let $e_1(n_1, n_2, T_T)$ be an untraversed edge incoming to n_1; then, n_2 must be not ended too (i.e., there is at least one not visited incoming edge within T). Thus, there is a path of unended nodes in $T' \cap T$; however, as there are no cycles, all nodes appear only once in the path. Thus, when the algorithm reaches the root (by definition there are no edges incoming to the root), all of its outgoing edges must have been traversed. This is a contradiction, given that, either there are no "not ended" nodes, or there is a cycle in the document.

We now study the complexity of Algorithm 3. Each node can be visited more than once, depending on the number of incoming edges. The best case is the one where no temporal cycles exist. In this case, lines 6 to 11 will never be executed. Lines 5 and 6 are of constant order, and the loop in line 17 is executed $deg_{out}(n)$ times; all operations are of constant order, resulting in order $deg_{out}(n)$. Lines 31 to 34 are also of constant order. The final loop is performed $|v|$ times in the worst case, and the operations are of constant order. Finally, we have: $\sum_{n \in |V|} deg_{out}(n) + |V| \approx O(|E| + |V|)$.

4.2 Fixing Inconsistent Documents

In the previous section we provided efficient algorithms that allow the user to quickly check if the document presents inconsistencies. In this section we will discuss how we can correct these inconsistencies. For each kind of inconsistency (of types i, ii and iii), we study possible fixing procedures. Of course, there are semantic implications for each of the solutions proposed here that the user must be aware of. If these implications are unacceptable for the user, she may just choose dropping the document instead of fixing it. We will study isolated inconsistencies, that is, we assume that everything happens as if the inconsistency under study is the only one in the document. The following definitions will be used in the remainder of this section.

Definition 8 (Deleting Edges in Temporal XML) Let D be a Temporal XML document, e be a containment edge of the form $e(n, m, T_e)$ and let $I = [I.FROM, I.TO]$ be a temporal interval. The deletion of e in the interval I is defined as follows:

1. If $I.FROM \leq T_e.FROM \leq T_e.TO \leq I.TO$, then physically delete e.
2. If $T_e.FROM < I.FROM \leq T_e.TO \leq I.TO$, then make $T_e.TO = I.FROM - 1$.

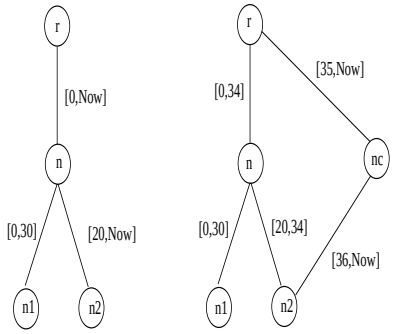

Fig. 5 Deleting the edge $(n, n_2, [20, Now])$ at t=35, using node duplication

3. If $I.FROM \leq T_e.FROM \leq I.TO < T_e.TO$, then make $T_e.FROM = I.TO + 1$.
4. If $T_e.FROM < I.FROM \leq I.TO < T_e.TO$, then
   (a) Create a new node n_c
   (b) Replace T_e in e by $[T_e.FROM, I.FROM - 1]$ and create a new edge $e'(n_c, m, [I.TO + 1, T_e.TO])$.
   (c) Remove every edge $e_j(n, n_j, T_{e_j})$ outgoing from n for which $I.FROM \leq T_{e_j}.FROM$ and create a new edge $e'_j(n_c, n_j, T_{e_j})$.
   (d) For every edge $e_j(n, n_j, T_{e_j})$ outgoing from n for which $T_{e_j}.FROM < I.FROM \leq T_{e_j}.TO$ replace T_{e_j} in e_j by $[T_{e_j}.FROM, I.FROM - 1]$ and create a new edge $e'_j(n_c, n_j, [I.FROM, T_{e_j}.TO)$.
   (e) Remove every edge $e_i(n_i, n, T_{e_i})$ incident to n for which $I.FROM \leq T_{e_i}.FROM$ and create a new edge $e'_i(n_i, n_c, T_{e_i})$.
   (f) For every edge $e_i(n_i, n, T_{e_i})$ incident to n such that $T_{e_i}.FROM < I.FROM \leq T_{e_i}.TO$ replace T_{e_i} in e_i by $[T_{e_i}.FROM, I.FROM - 1]$ and create a new edge $e'_i(n_i, n_c, [I.FROM, T_{e_i}.TO])$.

Note that step 4 in Definition 8 performs a duplication of the node from which the deleted edge outgoes. The next example illustrates the situation.

Example 5 Figure 5 shows a deletion of edge $e(n, n_2, [20, Now])$ at instant $t = 35$. Since $T_e.FROM < 35 < T_e.TO$, we performed node duplication (following step 4 in Definition 8) as follows: we created a copy of n, denoted n_c, and the edge $(n_c, n_2, [36, Now])$; we also replaced $(n, n_2, [20, Now])$ by $(n, n_2, [20, 34])$ (step 4(b)). According to step 4 (f), the edge $(r, n, [0, Now])$ has been replaced by $(r, n, [0, 34])$, and a new edge $(r, n_c, [35, Now])$ was created.

Definition 9 (Temporal Label Expansion and Reduction)

Given a containment edge $e(n_i, n_j, T_e)$, an expansion of T_e to an instant t is performed making $T_e.TO = t$, if $t > T_e.TO$, and $T_e.FROM = t$, if $t < T_e.FROM$.

Reducing the temporal label T_e to an interval $T' \subset T_e$ implies deleting e in the intervals $[T_e.FROM, T'.FROM - 1]$, $[T'.TO + 1, T_e.TO]$.

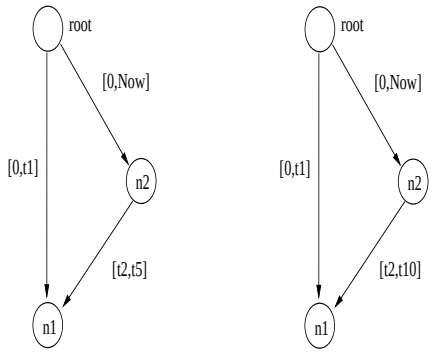

Fig. 6 Original graph, and graph after expansion at t_{10}

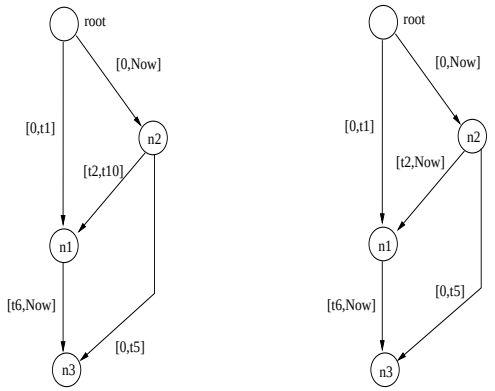

Fig. 7 Example of inconsistency of type i and solution by expansion

Figure 6 shows an example of expansion for the temporal label of the edge $e(n_2, n_1, [t_2, t_5])$ to instant t_{10}. In this case, t_{10} became the rightmost boundary of T_e of the edge between n_2 and n_1.

Definition 10 (Youngest (Oldest) Incoming Edge) We will denote *youngest edge incoming to a node* n, $y_e(n)$, an edge whose temporal label is the largest (according to the notation above) among all the temporal labels of the edges incoming to n. Analogously the *oldest edge incoming to a node* n, $o_e(n)$, is an edge whose temporal label precedes all the labels of the edges incoming to n.

Inconsistencies of Type i

In this case, the temporal label of an edge outgoing from a node is outside the lifespan of the node. We will say that an edge e is *inconsistent* if its temporal label is outside the lifespan of the *inconsistent node* (i.e. the origin node of e). For inconsistencies of type i, the interval of inconsistency I_I is the maximum interval within the temporal label of e that is not included in the lifespan of the inconsistent node. Note that I_I could actually be a set of intervals (for instance, if the lifespan of the inconsistent node is properly included in the temporal label of the inconsistent edge). In this section we will study the problems introduced by an inconsistent edge. We study two ways of fixing the problem: (a) Correction by expansion (expanding the lifespan of the inconsistent node); (b) Correction by reduction (reduces the temporal label of the inconsistent edge, closing I_I).

(a) *Correction by expansion.* In this solution, we expand the inconsistent node's lifespan until it covers the violating interval. We may take the youngest or oldest incoming edge, and modify its temporal label in a way such that it covers the whole label of the inconsistent edge. If $I_I \succ lifespan(n)$ then we must consider $y_e(n)$; if $lifespan(n) \succ I_I$, we must consider $o_e(n)$.

Example 6 Figure 7 shows that node n_1 presents an inconsistency of type i (the youngest edge incoming to n_3 has temporal label $[t_6, Now]$, and the lifespan of n_1 is $[0, t_{10}]$). Then, $I_I = [t_{11}, Now]$. The right hand side of the picture shows the solution, expanding the youngest edge incoming to n_1 i.e., $e(n_2, n_1, T)$.

Note that even though in Example 6 we only expanded one temporal label, this may not be the usual case: the modified interval may fall outside the lifespan of the origin node of the inconsistent edge. Thus, the inconsistency may recursively propagate upward in the path, until a consistent state is reached. To make these concepts more formal, we define the concepts of *path of youngest parents* and *path of oldest parents*. We will generically denote these paths *expansion paths*.

Definition 11 (Expansion Paths) We call *youngest parent of a node* n, the origin node of $y_e(n)$. Analogously, we denote *oldest parent of a node* the origin node of $o_e(n)$. A *path of oldest (youngest) parents* between two nodes n_i, n_j is a path where each node is the youngest (oldest) parent of the next node in the path. We denote these paths *expansion paths*.

It can be shown that all sub-paths of a path of youngest (oldest) parents are also paths of youngest (oldest) parents.

Example 7 For the graph in the right hand side of Figure 7, the path of youngest parents for node n_3 is $(n_3, n_1, n_2, root)$. The path of oldest parents for the same node is $(n_3, n_2, root)$.

The problem with the solution by expansion is twofold: on the one hand, we do not really know if the containment relation actually existed in the new interval. An expert user (or curator) will be needed to define this. On the other hand, the expansion may introduce a cycle (i.e., an inconsistency of type *iii*). In this case, expansion will not be a possible solution. We characterize the latter situation defining the *Instant of Maximal Path Expansion* (IMPE). The idea is that if we expand the interval beyond the IMPE, a cycle will be produced.

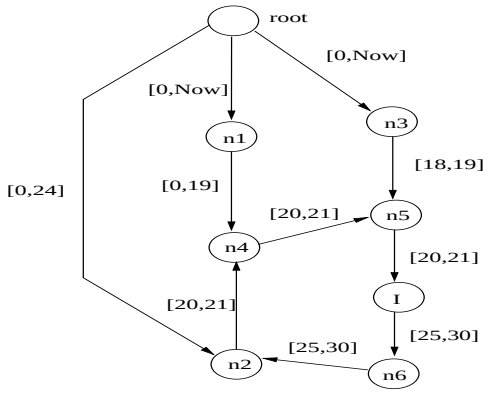

Fig. 8 Instant of Maximal Path Expansion

Definition 12 (Instant of Maximal Path Expansion (IMPE))
Let $P = (n_1, n_2, \ldots n_f)$ be an expansion path between n_1 and n_f, and let $e_i(n_i, n_{i+1}, T_i)$, with $1 \leq i \leq f-1$, be the edges in this path. Let $m = min\{T_1.TO, \ldots, T_{f-1}.TO\}$ and $M = max\{T_1.FROM, \ldots, T_{f-1}.FROM\}$. Let $L = [(n_f, \ldots, n_1, T'_1), \ldots, (n_f, \ldots, n_{f-1}, T'_{f-1})$ be a list of mcps such that $(n_f, \ldots, n_k, T'_k)$ is an mcp from node n_f to node n_k, with $1 \leq k \leq f-1$.

We define the *Instant of Maximal Path Expansion* of P, denoted $IMPE(P)$ as:

$$IMPE(P) = \begin{cases} \text{the maximum instant } t \text{ such that} \\ t \geq m \text{ and } [m,t] \cap T'_j = \phi \ \forall j \in 1..f-1, \\ \text{if } P \textit{ is a path of youngest parents.} \\ \\ \text{minimum instant } t \text{ such that} \\ t \leq M \text{ and } [t,M] \cap T'_j = \phi \ \forall j \in 1..f-1, \\ \text{if } P \textit{ is a path of oldest parents.} \end{cases}$$

The intuition behind this definition is that, in the case of a path of youngest parents for instance, the IMPE of a path $P = (n_1, ..., nf)$ is an instant greater than the minimum ending point of the intervals in an expansion path, and less than the starting point of the intervals of all mcps starting from a node reachable from n_f, and ending at node n_1.

Example 8 Figure 8 shows a graph with the expansion path (actually a path of youngest parents) (n_2, n_4, n_5). Node I violates consistency condition of type i. A solution for this could be to expand the lifespan of I. In this case, $IMPE(n_2, n_4, n_5) = 24$, because $t = 24$ is greater than the minimum ending time of the intervals in the expansion path, and less than the interval of the mcp between I and n_2. Thus, expanding to $t = 25$ would introduce a cycle. Then, the inconsistency cannot be solved by means of lifespan expansion.

Theorem 2 *Let D be a document with an inconsistence of type i in a node n. Then, the IMPE is the maximum instant to which an edge interval in an expansion path can be expanded without introducing a cycle in the document.*

Proof We will study the case of a path of youngest parents. (The case of a path of oldest parents is analogous.) Let us assume that we expand an interval until the IMPE, and a cycle is generated. Then, this implies that there is a path in some instant $t \in [min(T_i.TO), IMPE]$ (see Definition 12), between (a) n_f and n_i for some n_i in the path of youngest parents; (b) n_j and n_i for some n_i, n_j in the path of youngest parents. In case (a), this would imply $t \in T_{k_i}$, for some $mcp(n_f, n_k, T_{k_i})$, contradicting the definition of IMPE. In case (b), before the expansion, n_i and n_j were consistent before the expansion, thus, $t \in lifespan(n_i) \wedge t \in lifespan(n_j)$ holds, implying that the cycle was pre-existent.

(b) *Correction by reduction.* The main idea of this solution is to modify the temporal label of the inconsistent edge, in a way such that it lies within the lifespan of the starting node of such edge. It may even be necessary to delete this edge if its temporal label does not intersect the lifespan of the inconsistent node. Although not cycles can be introduced by this solution, it may introduce new inconsistencies of type i in the ending node of the modified edge if this node has outgoing edges that cover the interval that has to be reduced; moreover, inconsistencies of type ii may also be introduced if the deleted interval was not in one of the lifespan's extremes.

The algorithm for this solution proceeds as follows: it first deletes the edge *in the interval of inconsistency.* Then, it visits the node at the end of this edge and repeats the process until a consistent document is obtained. The number of iterations required by this solution is given by: $\sum_{n \in V} deg_{out}(n) \approx O(|E|)$.

Finally, in the worst case, inconsistencies of type ii must be fixed (with order $|E|^2$, see below), yielding an order of $O(|E| + |E|^2) \approx O(|E|^2)$.

Example 9 Figures 9 (a) and (b) show a graph where the *correction by reduction* approach generates new inconsistencies of type i and ii. In Figure 9 (a), reducing to [20,50] the interval of the edge (n_2, n_3) introduces a gap in node n_3. In Figure 9 (b), the same correction will make the temporal label of the edge (n_3, n_4) lie outside the lifespan of node n_3.

(c) *Expansion vs. Reduction* The discussion above showed that both fixing procedures, i.e., correction by expansion or correction by reduction may propagate upward or downward in cascade, respectively. For example, in the case of Figure 7, assume that the label of the edge $(root, n_2)$ is $[0, t_{10}]$ instead of $[0, Now]$. Fixing by expansion the inconsistency over n_1 as explained in Example 6, would propagate the inconsistency to node n_2.

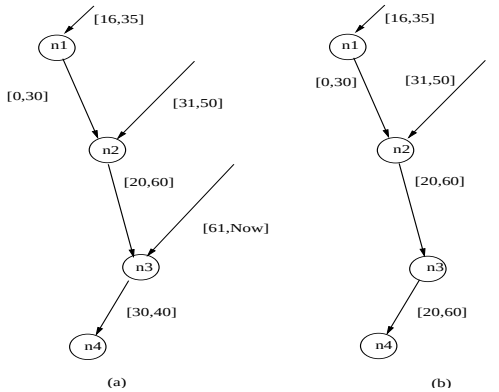

Fig. 9 Correction by reduction

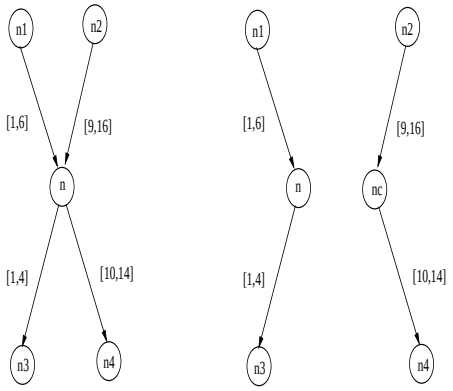

Fig. 10 Node duplication for fixing a gap inconsistency.

On the other hand, a correction by reduction may propagate downward and also introduce gaps (inconsistencies of type *ii*), as Example 9 shows. In order to compare options (a) and (b), a simple metric could be used, namely the number of changes needed to fix the problem, where a change could be: (a) the expansion of an interval; (b) the reduction of an interval; (c) the duplication of a node; (d) the deletion of an edge.

Inconsistencies of Type ii

As we already explained, these kinds of inconsistencies occur when some edges incoming to a node are not consecutive. This may be caused by: (a) overlapping of temporal labels, involving two or more of them; (b) the union of the temporal labels of the edges incoming to a node presents a temporal gap.

For fixing overlapping it suffices just to delete one of the violating edges in the interval of inconsistency. Fixing the gaps has more than one possible solution: (a) physically delete all incoming edges until the gap is closed; (b) expand the temporal labels of the edges, in order to close the gap (this could be performed expanding the temporal labels of one or more of the edges involved); (c) treat the inconsistency from a syntactic point of view, duplicating the violating node in a way such that the resulting incoming and outgoing edges have consistent temporal labels. This duplication is based on the same concepts underlying the fourth step of Definition 8.

The first two options have the following problem: they may introduce new inconsistencies of type i (for example, if the violating node is n, and there is an edge $e(n_i, n, T_e)$, and T_e is expanded to T'_e, the latter label may be outside the lifespan of n_i. Thus, we think the third option is the best one, if the node created is semantically equivalent and syntactically consistent. Figure 10 shows a gap inconsistency in node n fixed by node duplication at time instant 9. Note that in this case, duplication eliminates the gap between the time label of the edges incoming to n.

The algorithm for fixing inconsistencies of type *ii* visits all the document's nodes looking for gaps or overlapping. If an overlapping is found, one of the edges involved is deleted in the interval where the overlapping is produced. If a gap if found, the algorithm performing node duplication is called. Each time a node n is visited, the calling to the node duplication algorithm is performed $deg_{in}(n)$ times. This gives the algorithm an order of $O(|E|)^2$.

Inconsistencies of Type iii

Inconsistencies of type *iii* involve cycles occurring in some interval(s) of the document's lifespan. In this case, again, we have more than one possible way of fixing the inconsistency, basically consisting in deleting (according to Definition 8) edges within the cycle. We may:

- delete all containment edges involved in a cycle during the inconsistency interval I_I (i.e., in this case, the interval when the cycle occurs). This can be performed (a) by deleting (within I_I) all the subgraphs with root in each of the nodes in the cycle; or (b) by expanding the *expansion path* (see Definition 11) for each node belonging to the cycle.
- delete (within the interval of inconsistency) one of the edges in the cycle. Given that this would introduce an inconsistency of type i, this solution is only possible if there is at least one node n in the cycle with more than one incoming containment edge $e_c(n_i, n, T_e)$ such that T_e lies outside I_I. Thus, besides deleting the edge, T_e must be expanded in order to prevent introducing a new inconsistence.

Example 10 Figure 11 shows the two alternatives for fixing an inconsistency of type *iii*. In Figure 11(b) all edges involved in the cycle are deleted during $I_I = [0, 15]$. In Figure 11(c), the cycle was eliminated by only deleting the edge incoming to n_1 in the interval $[0, 15]$, and expanding the temporal label of the remaining edge incoming to n_1 (i.e., the label is now $[0, 35]$), in order to avoid an inconsistency of type i.

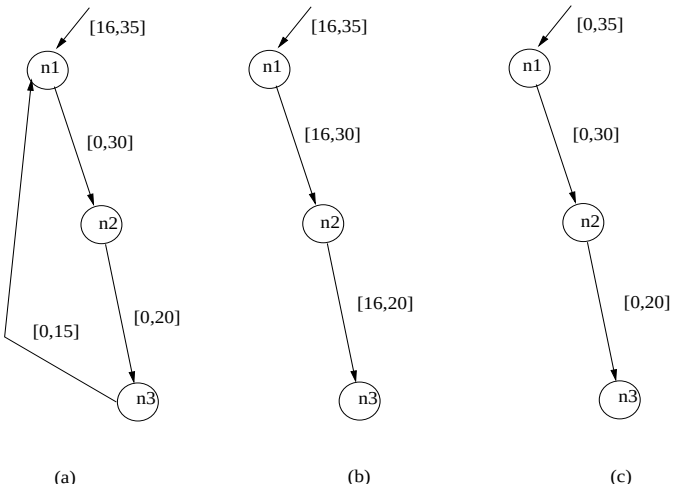

Fig. 11 Fixing an inconsistency of type *iii* (cycle).

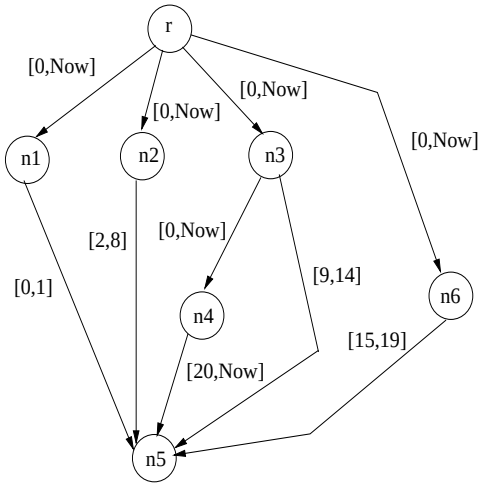

Fig. 12 Commutativity of gap elimination

Algorithm 4 performs cycle elimination by deleting all the edges within the interval of inconsistency.

Algorithm 4 (Fixing an Inconsistency of Type *iii*)
INPUT: A document D, with a cycle C in an interval T_c.
OUTPUT: a legal temporal XML document

Fixcycle(D, C, T_c) {
(1) n_c = *a node in* C
(2) *nodes_stack* = $[n_c]$
(3) **while** *node_stack is not empty* **do**
(4) n = *node_stack.pop*()
(5) *visited*$(n) = True$
(6) **for each** *edge* $e = (n, n_d, T_e)$ *outgoing from* n, $T_c \cap T_e \neq \phi$ **do**
(7) *delete* e *in* T_c
(8) **if** n_d *has no other incoming edges and* $T_e \subseteq T_c$ **then**
(9) *delete the subtree with root* n_d
(10) **else**
(11) **if** !*visited*(n_d)
(12) *node_stack.push*(n_d);
(13) **end if**
(14) **end for**
(15) **end while**
(16) Fix possible inconsistencies of type ii.
(17) return D }

Line (16) fixes all possible inconsistencies (basically gaps) that could have been introduced by a sequence of edge eliminations. Successive deletions of edges incoming to the same node may cause more than one gap when the labels of these edges were not at the beginning or the end of the node's lifespan. This would result in many node duplications. Thus, we decided to postpone node duplication to the end of the algorithm, because if the edges that are deleted have consecutive intervals the gaps could be solved in one single step (i.e., with just one node duplication). Figure 12 shows an example of this: the regular procedure for deleting edges (n_6, n_5), (n_2, n_5) and (n_3, n_5) implies three node duplications, in that order. Instead, if we just delete the edges and leave the action of fixing the gaps to be performed at the end of the whole process, we would have to perform just one node duplication and obtain the same end result (i.e., node n_5 and a copy of it, with intervals $[0, 1]$ and $[20, Now]$ respectively). The algorithm has an order $O(|E|^2)$ due to this last step.

The algorithm for eliminating a single edge in the cycle essentially picks a node n in the cycle such that n has at least another incoming edge with temporal label not in the cycles' interval T_c. The algorithm then deletes the edge incoming to n in T_c and expands (if possible, i.e., using the notion of IMPE introduced above) the lifespan of n including T_c in this lifespan, to avoid inconsistencies of type i.

5 Model Implementation

The abstract temporal model introduced in Section 3.2 can be encoded into a concrete XML document in many ways. We distinguish between non-replicated representations, where each node of the graph is represented by a single XML element or attribute, and replicated representations, where a node is represented by multiple elements or attributes. In the non-replicated representations, the nesting relationship of the resulting document is used to encode the "oldest" containment edges, while the remaining containment edges are represented by references. In the top-down version, the references go from parent to child, while in the bottom-up version they go from child to parent. Experiments we performed showed that the non-replicated representation outperforms the other ones in terms of space. Moreover, the replicated representations have some semantic issues we will briefly discuss. Thus, we will focus on the top-down non-replicated representation, which we will describe in detail in this section. For completeness of analysis we will give a quick idea of the other ones.

```
<NBAdb>
  <franchise ID="1" [0,Now]>
    <name [0,Now]> Raptors </name>
    <team [0,Now]>
      <player [0,20]>
        <name [0,20]> Oakley </name>
      </player>
      ...
      <player [0,20]ID="16">
        <name [0,Now] > McGrady </name>
        <stats [0,Now]>
           <goals[0,Now]>11</goals>
        </stats>
      </player>
      ....
  <franchise ID="2" [0,Now]>
    <name [0,Now]> Magic </name>
    <player [21,Now] IN ="16"/>
    ...
```

Fig. 13 Top-down non-replicated representation

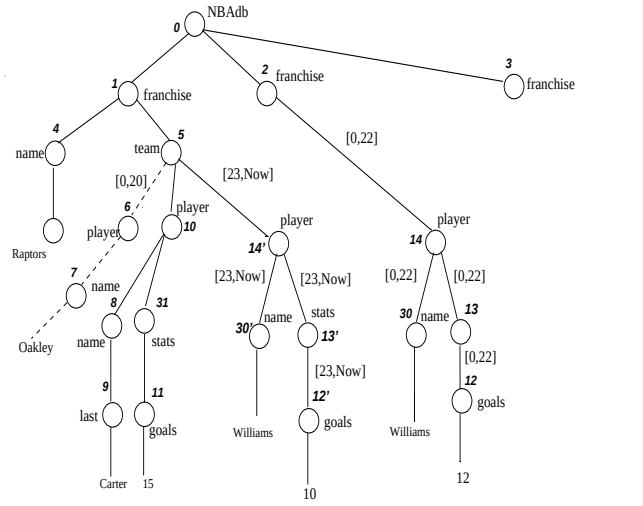

Fig. 14 Portion of the NBA database with duplicated nodes

5.1 Non-Replicated Representations

The non-replicated representation comes in two flavors: (a) Top down, and (b) Bottom-up.

(a) *Top-down.* The root of the graph maps to the root element of the document. For each element node there will be an element in the document, tagged with the label of the node. If the element node has a containment edge to a value node, the corresponding value is included in the element. For each attribute node there will be an attribute in the document, and its value will be the unique value node associated to the attribute node. If the attribute is of type REF, the value of the attribute will be the ID of the node being referenced.

Let $e(n_i, n_j, T_e)$ be one of the containment edges incoming to a node n_j. The element elem_{n_i} representing n_i in the XML document will physically include the element elem_{n_j}, tagged with the interval T_e. Thus, there will be only one element representing n_j in the document. For each node n_k in the remaining edges $e(n_k, n_j, T_{e_k})$ incoming to n_j, a distinguished reference attribute denoted IN with the value of the ID in elem_{n_j} and label T_{e_k} will be placed in the element elem_{n_k}.

The containment edges to be physically encoded in the XML document can be selected in many different ways. In general, we can chose a time instant t and for each containment edge $e(n_i, n_j, T_e)$ such that $t \in T_e$, physically include n_j in n_i (this is equivalent to taking a snapshot of the graph at time t and generate the XML document representing this snapshot); other containment edges incoming to n_j (if they exist) will be referenced as explained above. All nodes n_j such that t is not included in T_e, must be added afterward. As another alternative, we could take a different time instant t_j for each node n_j and physically include n_j in n_i if there is a containment edge $e(n_i, n_j, T_e)$ and $t_j \in T_e$. Following this approach, in the work presented here we physically encoded the "oldest" containment edges.

Example 11 For the sake of clarity, in the following examples we will use a simplified syntax for the XML documents resulting from the various mappings. For instance, we use the notation <franchise ID='1'[0,Now]> to mean that the time interval associated with this element is [0,Now]. (Note that we use integers to represent time points instead of actual date/time values, also for simplicity). In an actual implementation, we define a namespace and create three new attributes: 'FROM'(the starting point of the interval), 'TO'(the ending point of the interval), and 'IN' (the reference to a contained element). In Section 5.4 we describe the implementation of temporal features in more detail.

Figure 13 depicts a portion of the document resulting from mapping the graph in Figure 1. Here, attributes of ID type have no temporal tag. Let us consider the second `player` element, with temporal interval [0,20]. The "oldest" containment edge approach has been chosen, resulting in the inclusion of this player in the `<team>` element corresponding to the Toronto Raptors. The construct <`player[21,`Now`] IN='16'/`> represents a current containment edge going from node '2' to node '16' with time label [21,Now]. This means that the information about this player is physically encoded in the element with ID = '16'.

Bottom-up. In the top-down representation we picked the oldest containment edge to be represented by physical inclusion, while the others were represented by references from parent to child. We could instead have the references going from child to parent. For example, instead of placing a reference to node '16' between instants '21' and 'Now', we place a reference from the player to his current franchise. The resulting document is analogous to the one obtained adopting the top-down alternative.

5.2 Node-Replicating Representation

A third alternative to the representation described above avoids using the 'IN' reference. This implementation requires transforming the original graph into a tree of containment edges. This is performed, in short, recursively creating k copies of each node n with k ($k > 1$) incoming containment edges. This process is similar to the one described in Definition 8 and transforms the temporal XML document into a tree of containment edges. Figure 14 shows a portion of the graph for the NBA database with node replication, where the node for player with ID='14' has been duplicated, denoting '14′' the new ID. As a convention, for each node with node number n that is duplicated, we denote its new versions n'. Note that the lifespan of node 12 (the interval $[0, Now]$) has been split into intervals $[0, 22]$ and $[23, Now]$. This shows the biggest weakness of this approach: node replication is not appropriate when the value nodes contain data that aggregates over time. In Figure 14, we have assigned values 12 and 10 to nodes 12 and is replica, respectively, assuming that the value associated to the original node (22 goals in this case) is partitioned proportionally to the lifespan of the nodes involved in the replication.

5.3 Node-Edge Representation

A fourth way of implementing a temporal XML document is to store the edges and nodes of the graph in a way similar to the *edge XML-to-relational mapping* [21]. The idea is to list the nodes and the edges in the graph, using attributes for their validity intervals and other features like references or attributes. For instance, there are two elements, `NODE` and `EDGE`, that define the nodes and the edges respectively. Additionally, there are two attributes, `Origin` and `End` (defined within a namespace), that represent the node numbers that are the endpoints of each edge. Finally, a `Type` attribute defines the type of the node being represented (i.e. element, attribute or value nodes).

5.4 Implementation of Temporal Attributes

As we commented above, the syntax for intervals and distinguished references introduced in the temporal document was simplified for the sake of the paper's clarity. In a real implementation, we need to define a namespace and create three new attributes: 'FROM'(the starting point of the interval),'TO'(the ending point of the interval), and 'IN' (the reference to a contained element). We denote this namespace 'Time', and its associated URI is defined as 'http://www.cs.toronto.edu/ db/time'. Attributes 'Time:FROM' and 'Time:TO' introduce potential attribute duplication in a tag. Thus, we adopted the solution explained in Section 3. Figure 15 shows an example. Note that for references of type `IN`, we defined

```
<NBAdb xmlns:Time="hhtp://www.cs.toronto.edu/db/time">
 <franchise ID="1" Time:FROM="1999-01-01"
 Time:TO="Now">
 ...
   <team  Time:FROM="1999-01-01" Time:TO="Now">
    <player Time:FROM="1999-01-01"
    Time:TO="2001-06-01" Time:IN="7">
    ...
    <player Time:FROM="1999-01-01"
    Time:TO="2000-12-31" Time:IN="3">
    </player>
      <ATTRIBUTES>
      <day-of-birth Time:FROM="2002-01-01"
      Time:TO="Now" $>$ "5-24-79" </day-of-birth>
      </ATTRIBUTES>
         <assists Time:FROM="1999-01-01"
         Time:TO="2000-05-31">6.5</assists>
         <assists Time:FROM="1999-01-01"
         Time:TO="2000-05-31">6.5</assists>
     ...
```

Fig. 15 Implementation of temporal attributes

a special attribute 'Time:IN', included in the tags of the types of the element being referenced (see for example, tags <`team`> and <`player`>).

5.5 Snapshots

In temporal relational databases it is often relevant to compute *snapshots* of the data. In temporal XML we would like to be able to reconstruct a document as of a given time instant. We call this a *document snapshot.* In Section 6 we will distinguish this concept from the notion of *snapshot query.* In this section we briefly show how to compute a document snapshot (at time t) of a temporal document implemented as described in Sections 5.1 (using the top-down alternative) and 5.2.

Snapshots in a Non-Replicated Implementation The following procedure computes a document snapshot as of a time instant t, when the temporal document is implemented using the top-down non-replicated implementation.

- There is a non-annotated tag T in $\mathcal{D}(t)$ for every tag T in $\mathcal{D}$ annotated with a temporal element T_e where: (a) $t \in T_e$; (b) T is not contained in any tag T_1 such that there exists a distinguished reference of type IN to T_1 at time t (see Example 12).
- For every reference r annotated with a temporal element T_e such that $t \in T_e$ included in an element satisfying condition (b) above, there is a non-annotated reference r in $\mathcal{D}(t)$.
- For every attribute a=v (where v is the value associated to a) annotated with a temporal element T_e such that $t \in T_e$ included in an element satisfying condition (b) above, there is a non-annotated attribute a in $\mathcal{D}(t)$.

```
<NBAdb>
  <franchise ID="1">
    <name> Raptors </name>
    <team>
      ...
  <franchise ID="2">
    <name> Magic </name>
    <player ID="16">
       <name>McGrady</name>
       <stats>
         <goals>11</goals>
       <stats>
    </player>
     ...
```

Fig. 16 Snapshot of the document of Figure 13 at t='24'

- If within a tag T in $\mathcal{D}$ there is a tag T_1 with a reference R of the form IN=v to an element with ID=v (also with tag T_1) such that for the temporal element labeling R, call it T_r, $t \in T_r$ holds, include in $\mathcal{D}(t)$ the complete element being referenced (i.e., T_1) as the last subelement within tag T, excluding the sub-elements with temporal labels T_i where $t \notin T_r$. Finally, replace IN=v with ID=v in T_1.
- The former are the only transformation rules applying to the document.

Example 12 We will give an example of the procedure above, using the document in Figure 13. When taking a snapshot at time '24', the tag <`player[0,20] ID='16'...`> neither verifies condition (a), nor condition (b). The tag <`name[0,`Now`]`> `McGrady` </`name`> verifies condition (a) but not condition (b), which prevents its inclusion in the snapshot (inside the `player` tag). However, notice that there is a tag <`player`[$21, Now$] `IN='16'...`> (T_1 in the fourth step above), included in the franchise tag with ID='2' (tag T above). Thus, because of the fourth step of the algorithm, the snapshot will have an element <`player ID='16'..`>. Also, all the sub-elements of <`player[0,20] ID='16'..`> will be included in the <`franchise`> tag (note that all the sub-elements are labeled $[0, Now]$ in the document, and that a snapshot does not have temporal labels). A portion of the resulting snapshot is shown in Figure 16.

Snapshots in a Node-Replicating Implementation Taking a snapshot at time t of a temporal document implemented as described in Section 5.2 just requires scanning the document and placing a tag for every tag in $\mathcal{D}$ annotated with a temporal element T_e such that $t \in T_e$. Analogously, an attribute and/or reference must be created for each attribute and/or reference associated with a temporal element including t.

The node-replicating implementation requires only one pass through the document for computing a document snapshot, while the non-replicated implementation requires at least two, the first one for finding the references, and the second one for generating the snapshot. This is compensated by the size of the produced document, which, due to node duplication, is two to three times larger than an equivalent document with no duplicates. As a result, computing a document snapshot in both representations takes, on the average, approximately the same time. In Section 10 we provide experimental results on snapshot computation.

6 TXPath: a Temporal Extension to XPath

One of the motivations for proposing a temporal data model is to support query languages that make complex queries easy to express. For example, consider the query "players who played for the Toronto Raptors continuously since at least the year 2000." In this section we introduce TXPath, a temporal query language that extends XPath 2.0 [60] with temporal operators in order to enable this kind of query. As we only intend to show the main ideas of this extension, we will not discuss details or standard temporal database issues like temporal comparisons and granularity, that are treated in the usual way.

6.1 Syntax and Semantics

In non-temporal XPath 2.0, the meaning of a path expression is the sequence of nodes, at the end of each path, that matches the expression. In TXPath, the meaning is a sequence of (node,interval) pairs such that the node has been at the end of a matching path continuously during that interval (i.e., at the end of a *continuous path*).

We stay as close as possible to the XPath syntax, extending it with temporal operators. We specify the TXPath semantics adapting the formal XPath semantics introduced by Wadler [54]. The meaning of an XPath expression is specified with respect to a *context node*; we extend this to a *context pair* of a node and a time interval. We define three semantic functions: $\mathcal{S}, \mathcal{Q}$ and $\mathcal{Q}_T$ such that $\mathcal{S}[\![p]\!]x$ denotes the sequence of pairs $(node, interval)$ (or values, as we will see below) selected by pattern p when x is the context pair. The boolean expression $\mathcal{Q}[\![q]\!]x$ denotes whether or not the qualifier q is satisfied when the context pair $(node, interval)$ is x. Finally, another boolean expression $\mathcal{Q}_T[\![q_T]\!]x$ denotes whether or not a temporal condition q_T is satisfied. For the sake of brevity, in Figure 17 we only show the most common TXPath constructs.

Example 13 The expression `//player`, applied to the document in Figure 3, will return the sequence $(p1, [95, Now])$, $(p2, [99, Now])$, $(p3, [98, Now])$).

With respect to our running example, the query "players who have played for the Toronto Raptors continuously since the year 2000" reads in TXPath:

```
//franchise[name='Raptors']//player[@from≤2000
```

$$
\begin{array}{lcl}
\mathcal{S}[\![/p]\!]x & = & \mathcal{S}[\![p]\!]root(x)\ ; \\
\mathcal{S}[\![//p]\!]x & = & \{x_2 \mid x_1 \in subnodes(root(x)), x_2 \in \mathcal{S}[\![p]\!]x_1\ \}; \\
\mathcal{S}[\![p_1/p_2]\!]x & = & \{(v_2, I_1 \cap I_2) | (v_1, I_1) \in \mathcal{S}[\![p_1]\!]x, (v_2, I_2) \in \mathcal{S}[\![p_2]\!](v_1, I_1)\ \}; \\
\mathcal{S}[\![p_1//p_2]\!]x & = & \{x_2 \mid x_1 \in subnodes(x), x_2 \in \mathcal{S}[\![p]\!]x_1\ \}; \\
\mathcal{S}[\![p[q]]\!]x & = & \{(v, I) | (v, I) \in \mathcal{S}[\![p]\!]x, \mathcal{Q}[\![q]\!](v, I)\ \}; \\
\mathcal{S}[\![n]\!]x & = & \{(v, I) \mid isElement(v), child(x) = (v, I), name(v) = n\ \}; \\
\mathcal{S}[\![@n]\!]x & = & \{(v, I) \mid isAttribute(v), child(x) = (v, I), name(v) = n\ \}; \\
\mathcal{S}[\![@\mathbf{from}]\!]x & = & \{f \mid (v, I) \in \mathcal{S}[\![p]\!]x, I = [f, t]\ \}; \\
\mathcal{S}[\![@\mathbf{to}]\!]x & = & \{t \mid (v, I) \in \mathcal{S}[\![p]\!]x, I = [f, t]\ \}; \\
\mathcal{S}[\![\ p[q_T]\ ]\!]x & = & \{(v, I) \mid (v, I) \in \mathcal{S}[\![p]\!]x, \mathcal{Q}_T[\![p]\!](v, I)\ \}; \\
\mathcal{S}[\![ancestor :: p]\!]x & = & \{x_2 \mid x_1 \in prenodes(x), x_2 \in \mathcal{S}[\![p]\!]x_1\ \};
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{Q}[\![p = s]\!]x & = & \{(v, I) \mid (v, I) \in \mathcal{S}[\![p]\!]x, value(v) = s\} \neq \phi; \\
\mathcal{Q}[\![p]\!]x & = & \{x_1 \mid x_1 \in \mathcal{S}[\![p]\!]x\} \neq \phi;
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{Q}_T[\![d\ \texttt{IN}\ (@\mathbf{from}, @\mathbf{to})]\!]x & = & \{x \mid x = (v, [@\mathbf{from}, @\mathbf{to}]), d \geq @\mathbf{from}, d \leq @\mathbf{to}\} \neq \phi; \\
\mathcal{Q}_T[\![(s, e)\ \texttt{CONTAINS}\ (\mathbf{from}, @\mathbf{to})]\!]x & = & \{x \mid x = (v, [@\mathbf{from}, @\mathbf{to}]), s \leq @\mathbf{from}, e \geq @\mathbf{to}\} \neq \phi; \\
\mathcal{Q}_T[\![(s, e)\ \texttt{MEETS}\ (\mathbf{from}, @\mathbf{to})]\!]x & = & \{x \mid x = (v, [\mathbf{from}, @\mathbf{to}]), [\mathbf{from}, @\mathbf{to}] \cap [s, e] \neq \phi\} \neq \phi; \\
\mathcal{Q}_T[\![\ @\mathbf{from}\ \texttt{op}\ d]\!]x & = & \{x \mid r \in \mathcal{S}[\![@\mathbf{from}]\!]x, r\ \texttt{op}\ d\} \neq \phi; \\
\mathcal{Q}_T[\![\ @\mathbf{to}\ \texttt{op}\ d]\!]x & = & \{x \mid r \in \mathcal{S}[\![@\mathbf{to}]\!]x, r\ \texttt{op}\ d\} \neq \phi;
\end{array}
$$

Where $subnodes(y) = \{(v, I) \mid$ there exists a *maximal continuous path* (*mcp*) from y to v with interval $I\}$; $prenodes(y) = \{(v, I) \mid$ there exists an *mcp* from v to y with interval $I\}$; $root(x)$ is the $(root, interval)$ pair of the tree in which x is a $(node, interval)$ pair; $child(x) = \{(v, I) \mid$ there exists an *mcp* of length 1 from x to v with interval $I\}$.

Fig. 17 Formal semantics of TXPath

```
and @to='Now']
```

We use the XPath construct `@from` to refer to the starting point of the interval associated with each node in the answer, and similarly for `@to`.

6.2 TXPath by Example

We will briefly present and discuss the main features of TXPath, in order to give the idea of the kinds of queries that can be supported.

Coalescing Sequences In temporal queries it is often useful to coalesce sets of overlapping intervals. We define the *coalesce* operation over a sequence of pairs ($value(node)$, $interval$), where $value(node)$ stands for the value associated to a value node, to generate a new sequence where all maximal sets of overlapping intervals are coalesced into single intervals when the values are the same. For example, given a sequence S=((2,[1,5]), (2,[3,8]), (4,[12,16]), (4,[14,18])), `coalesce(S)` returns the sequence ((2,[1,8]), (4,[12,18])). Given an arbitrary sequence of pairs, we extend the XPath `distinct-values` operator to group all pairs that have the same node component and coalesce the resulting sub-sequence. For example, the query "goals scored by Carter whenever a change in his scoring occurred" is expressed as

```
distinct-values(//player[name='Carter']//goals)
```

This query only returns one pair $(goal, interval)$ for each sequence of k consecutive or overlapping seasons where Carter scored the same number of goals, instead of the k pairs that would be returned without using the `distinct-values` statement.

Aggregation XPath 2.0 has aggregate operators that can be applied to a sequence of nodes to compute its sum, average, etc. In addition, we can take advantage of these operators by applying them to sequences of time points, as the next example shows. The query "name of the players who were with the Orlando Magic when McGrady joined the franchise for the first time" is expressed in TXPath as:

```
let $m= min(//franchise[name='Magic']//
    player[name='McGrady']/@from)
return
   //franchise[name='Magic']//player[$m ≥ @from
     and $m ≤ @to]/name
```

In the query above, `min` returns the minimum time instant in the result set, and this value is used to qualify the results in the next part of the query.

Snapshots In Section 5 we discussed document snapshots based on the different implementations of the abstract temporal data model. Obtaining a document snapshot means reconstructing a temporal XML document as of a given time instant. In order to express document snapshots in TXPath we would need to introduce user-defined functions like the ones supported in XQuery. On the other hand, snapshot queries can be expressed in TXPath within the framework given by the syntax and formal semantics introduced in Section 6.1. A snapshot query at a time instant t is simply a query that retrieves a portion of a document as of t. The query returns a

sequence of pairs $(node, interval)$ such that the interval contains the instant t.

An example of a snapshot query is *"Give me the player nodes for players with the Toronto Raptors on October 10th, 2001"*. This query reads in TXPath:

```
NBAdb/franchise[name='Raptors']//players
   [@from ≥ '10/10/01' and @to ≤ '10/10/01']
```

Assuming that the date October 10th, 2001 is represented by instant 15, the result of this query over the database of Figure 1 is the sequence $((6, [0, 20]), (10, [0, Now]), (16, [0, 20]))$.

6.3 The Notion of "Now"

It is a well-known fact in temporal databases, that using a current time variable has several implications which require the definition of a precise semantics [12]. Since our model follows the transaction time approach, the problems arising from the use of *Now* are considerably reduced compared with a valid time data model. The main reason for this is that in valid time databases timestamps are provided by the user, while in transaction time databases these values are usually built-in, i.e., provided by the database management system (DBMS).

The semantics adopted for *Now* in this work is the one proposed in [12]. Therefore, the meaning of the current time variable is that, if the ending point of a temporal label is $T.TO = Now$, the edge is valid from $T.TO$ (the starting point of the label) until the timestamped element is updated, yielding the so-called *until changed* semantics. This will become evident in Section 9, where we discuss updates. A direct consequence of this decision is that $T.FROM$ can never be stamped with *Now*, as it could be the case in valid time databases.

From the language point of view, we have decided that the syntax for representing the current time variable uses the distinguished constant 'Now'. At the implementation level, for simplicity we have defined the `maxint` value for representing the end of time (some systems use '999-12-31' for representing this value). Also, a *current-date* function is applied when needed, that is, when 'Now' is found in a query, or `maxint` is found in the database.

7 Structural Summaries For Temporal XML

As we mentioned in Section 1, efficiently querying temporal XML documents requires the ability to find the paths in the graph that were valid at a given time (*i.e.* the *continuous paths* in the document). This ability is not provided by traditional path summaries. Our proposal for a new class of summaries, which we call *TSummary*, adds the time dimension to the usual path summarization by considering *continuous paths* to element or value nodes. TSummary is the theoretical framework behind TempIndex, our indexing scheme for temporal XML data [37].

7.1 Summarizing Continuous Paths

Structural summaries are data structures used to locate specific fragments of the XML data, such as nodes, paths and subtrees. By accessing relevant data directly they help to avoid sequential scans of entire documents during query evaluation. Since our goal is to optimize TXPath query evaluation, the (temporal) XML fragments we want to summarize are continuous paths. A TSummary includes a graph that describes the continuous paths in the temporal document in a concise way. Nodes in the temporal document are partitioned into equivalence classes. Each node in a TSummary graph will have associated to it one such equivalence class, which we call the *temporal mapping* (or *tmap*) of the summary node.

Like in non-temporal XML, a concise representation of the nodes based on their labels is a useful summarization of the temporal XML graph structure. The first TSummary we will present in this work is the *LCP summary* which summarizes labels of *continuous paths* from the root. Traditional path indices [25,38,31] often define equivalence classes of nodes that belong to paths with the same label. In contrast, the LCP Summary defines equivalence classes of nodes that belong to cp's from the root with the same label. Since we also need to summarize *temporal intervals*, we will define a summary that describes cp's regardless of their labels. One way of doing that is to cluster together nodes that belong to cp's from the root with the same length. This is in fact the definition of another TSummary, the *interval summary*.

We introduce next our formalization of temporal summaries we will use in the remainder of the paper.

Definition 13 (Temporal Summary) Consider a temporal XML document D and the set $TNode$ of pairs $\langle n, I\rangle$ such that n is a node in D, I is an interval, and there is a continuous path p from the root of D to n with interval I. A *temporal summary of* D, $S_D = (TSum, tmap, edge, Label, \lambda)$, is a structure where

- $TSum$ is a set of summary nodes defined as follows: $Sum = \{s \mid \langle s, n, I\rangle \in tmap\}$;
- $tmap$ is a relation defined as follows:
  - Each pair $\langle n, I\rangle$ is associated to only one summary node. That is, $\langle s, n, I\rangle \in tmap \Rightarrow \neg\exists s' \neq s \mid \langle s', n, I\rangle \in tmap$;
  - Every document node is associated to some summary node. That is, $\forall\langle n, I\rangle \in TNode : \exists s \in TSum \mid \langle s, n, I\rangle \in tmap$;

  We say that a pair $\langle n, I\rangle \in TNode$ is in the *temporal map* of a node $s \in TSum$ iff $\langle s, n, I\rangle \in tmap$.

– *edge* is a relation in $TSum \times TSum$ that represents the edges in S_D;
– λ is a labelling function that assigns names to nodes in *TSum* by mapping $TSum \rightarrow Label$.

Definition 14 (Temporal Summary Graph) Consider a temporal summary $S_D = (TSum, tmap, edge, Label, \lambda)$. The tuple $G_S = (TSum, edge, Label, \lambda)$ is the *summary graph* of S_D.

Since we need for the edge structure of the summary to somehow *describe* the structure of the temporal document, there has to be a relationship between the summary edges and the temporal XML graph edges. This relationship is given by the following property.

Property 1 (Edge Property) A temporal summary S_D has the *edge property* iff its edges are defined by *edge* as follows: $edge := \{\langle s, s' \rangle \mid \exists \langle s, n, I \rangle \in tmap \wedge \exists \langle s', n', I' \rangle \in tmap \ \wedge \ \exists e_c(n, n', I_e) \in D\}$

That is, there is an edge between two nodes in the summary iff there is a containment edge between any two nodes in their temporal mappings.

In order to define our first TSummary, we will need the notion of label of a continuous path. The standard notion of label paths can be easily extended to continuous paths as follows. Let $p = (n_1, \ldots, n_k, T)$ be a continuous path with interval T. The label path of p, or *continuous label path* $\lambda(p)$ is the concatenation of the labels of the n_i in p.

We are able to introduce now the *LCP summary*, a TSummary that summarizes labels of continuous paths from the root.

Definition 15 (LCP Summary) Consider a temporal XML document D and the set $TNode$ of pairs $\langle n, I \rangle$ such that n is a node in D, I is an interval, and there is a continuous path p from the root of D to n with interval I. A *temporal summary of D* $S_D = (TSum, tmap, edge, Label, \lambda)$ is an *LCP summary* iff

– S_D has the *edge property*;
– Two document nodes belong to the *tmap* of the same summary node iff they are at the end of continuous paths from the root with the same label : $\forall \ \langle n, I \rangle, \langle n', I' \rangle \in TNode, s \in TSum : \langle s, n, I \rangle, \langle s, n', I' \rangle \in tmap \Leftrightarrow \{\lambda(p) \mid p = (r, \ldots, n, I)\} = \{\lambda(p') \mid p' = (r, \ldots, n', I')\}$.
– *Label* is the set of node labels in the temporal document;
– $\forall \langle s, n, I \rangle \in tmap : \lambda(s) := \lambda(n)$.

Note that *tmap* defines a partition of $TNode$ where two pairs belong to the same equivalence class iff they have incoming continuous paths with the same labels. Also note that there is a one-to-one mapping between the equivalence classes defined by *tmap* and the summary nodes in $TSum$.

The following example shows the LCP summary for the NBA database fragment of Figure 1.

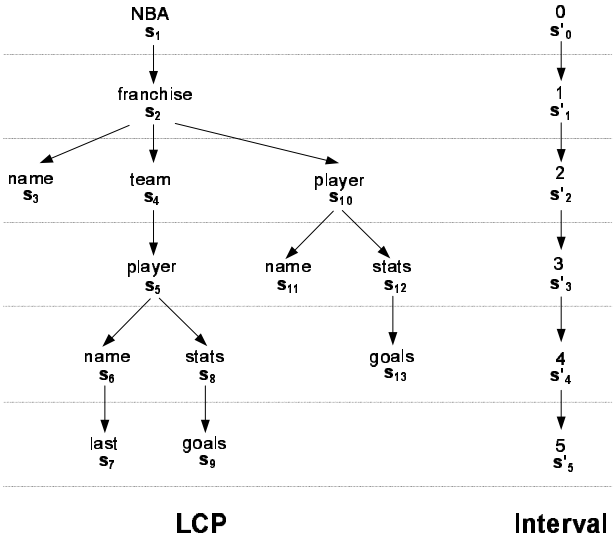

Fig. 18 Temporal Summary Graphs

Example 14 The LCP summary graph $G_S = (TSum, edge, Label, \lambda)$ for the NBA database is shown on the left side of Figure 18, where $TSum = \{s_1, \ldots, s_{13}\}$, *edge* is defined by the edges in the figure, $Label = \{$NBA, franchise, name, last, team, player, stats, goals$\}$, and λ is defined by the label assigned to each node in the figure. In addition, the *tmap* relation of the LCP summary S_D is given by the following table:

tmap		*tmap* (cont.)	
s_1	$0, [0, Now]$	s_8	$31, [0, Now]$
s_2	$1, [0, Now]$	s_9	$11, [0, Now]$
s_2	$2, [0, Now]$	s_9	$12, [23, Now]$
s_2	$3, [0, Now]$	s_9	$19, [0, 20]$
s_3	$4, [0, Now]$	s_{10}	$14, [0, 22]$
s_3	$15, [0, Now]$	s_{10}	$16, [21, Now]$
s_3	$25, [0, Now]$	s_{10}	$24, [0, Now]$
s_4	$5, [0, Now]$	s_{11}	$17, [21, Now]$
s_5	$6, [0, 20]$	s_{11}	$23, [0, 20]$
s_5	$10, [0, Now]$	s_{11}	$30, [0, 22]$
s_5	$14, [23, Now]$	s_{12}	$13, [0, 22]$
s_5	$16, [0, 20]$	s_{12}	$18, [21, Now]$
s_6	$7, [0, 20]$	s_{12}	$22, [0, Now]$
s_6	$8, [0, Now]$	s_{13}	$12, [0, 15]$
s_6	$17, [0, 20]$	s_{13}	$12, [16, 22]$
s_6	$30, [23, Now]$	s_{13}	$19, [21, 30]$
s_7	$9, [0, Now]$	s_{13}	$19, [31, Now]$
s_8	$13, [23, Now]$	s_{13}	$20, [0, 10]$
s_8	$18, [0, 20]$	s_{13}	$21, [16, Now]$

Since cp's in the LCP summary are clustered by label, we need additional summaries to describe the intervals and to capture the node ordering $<_t$ at any given instant (as defined in Proposition 1). In order to do that, we will introduce next a TSummary based on the notion of *temporal depth*.

Definition 16 (Temporal Depth) Consider a temporal XML document D. For each node n in D such that there exists a continuous path $p = (r, \ldots, n, I)$ in

D, $\delta(n, I) = length(p)$ is a function called the *temporal depth* of n during the interval I. (Note that there is at most one continuous path with interval I from the root to each node n).

For each temporal depth k, we define the nodes that are valid at that depth during an interval I as follows.

Definition 17 (Node Validity) A node n is *valid* at temporal depth k in an interval I iff there exists an interval I' such that $\delta(n, I') = k$ and $I \subseteq I'$.

Based on the notions of temporal depth and node validity we introduce next the *interval summary*, a TSummary that clusters together nodes that belong to cp's from the root with the same length.

Definition 18 (Interval Summary) Consider a temporal XML document D and the set $TNode$ of pairs $\langle n, I\rangle$ such that n is a node in D, I is an interval, and there is a continuous path p from the root of D to n with interval I. A *temporal summary of* D, $S_D = (TSum, tmap, edge, Label, \lambda)$ is an *interval summary* iff

- S_D has the *edge property*;
- Two document nodes belong to the *tmap* of the same summary node iff they have the same temporal depth: $\forall\ \langle n, I\rangle,\ \langle n', I'\rangle \in TNode, s \in TSum : \langle s, n, I\rangle, \langle s, n', I'\rangle \in tmap \Leftrightarrow \delta(n, I) = \delta(n', I')$.
- $Label = \{0, \ldots, m\}$, where m is the length of the longest cp in the document;
- $\forall\langle s, n, I\rangle \in tmap : \lambda(s) := \delta(n, I)$.

Note that $\delta(n, I)$ defines an equivalence relation between the nodes in the temporal XML graph where for each pair $\langle n, I\rangle$ in a class the length of the continuous path from the root to n is the same.

Example 15 The interval summary graph $G_{S'} = (TSum', edge', Label', \lambda')$ for the NBA database is shown on the right side of Figure 18. Since the difference in labels does not matter here, several LCP summary nodes may "collapse" into one in the interval summary. For instance, nodes s_3, s_4 and s_{10} of the LCP summary are represented by node s'_2 in the interval summary. This also impacts on the definition of $tmap'$ for the interval summary S'_D: all pairs $\langle n, I\rangle$ that belong to nodes s_3, s_4 and s_{10} in S_D (see *tmap* definition in Example 14) belong to node s'_2 in S'_D.

Whereas the LCP summary provides a combined label path + temporal clustering, the interval summary is in fact a pure temporal clustering. This kind of clustering does not consider node labels or label paths and therefore can be used for efficiently selecting nodes based solely on their intervals. This functionality is useful for computing document snapshots and for some stages in the evaluation of TXPath queries (see Section 8.2).

There are many proposals in the literature for indexing temporal intervals. Some of them are based on the methods proposed by Bozkaya *et al.* [3] and Salzberg *et al.* [47], where a B+ tree indexes the `FROM` value in the intervals being indexed, and each internal node is augmented with the information of the maximum `TO` value in an interval of the corresponding subtree. These proposals are "indexing" schemes rather than summaries. That is, they provide low level index structures and access methods for optimization. In contrast, TSummaries are high level descriptions of the temporal data which are in turn implemented by indexing schemes. In the next section we provide a description of our own indexing scheme, TempIndex, but TSummaries could also be implemented by combining other well-known interval indexes and access methods like the ones mentioned above.

7.2 TempIndex: an Indexing Scheme for Temporal XML

In order to optimize TXPath query evaluation, we need to integrate LCP and interval summaries in an effective way. In addition to the summaries themselves, we need indexes, access methods and additional data structures with information about the hierarchical relationships between nodes in a temporal XML documents. We present here TempIndex (introduced in previous work [37]), an indexing scheme that integrates LCP and interval summaries with additional indexes for efficient navigation.

For representing the structural relationship between nodes in different equivalence classes we define what we call *CP tables*. Each CP table is associated to a summary edge and stores the parent-child relationship between document nodes in the two equivalence classes of the end-points of such edge. In addition, CP tables contain the interval of the continuous paths ending at the child equivalence class. The information contained in the CP tables is used during query evaluation to traverse continuous paths with a given label and interval (see Section 8 for more details).

Definition 19 (CP Tables) Consider the summary S_D of a temporal XML document D. For each edge $e = (s_1, s_2)$ in the temporal summary graph G_S there is a *CP table* in which each tuple `t` has attributes `parent`, `node`, `from` and `to` such that there is a continuous path from the root of d to `t.node` with interval `[t.from, t.to]` via `t.parent`. When `t.node` has a value v associated to it, the CP table has an extra attribute named `value`, where `t.value`$= v$. Tuples in the CP tables are sorted by `node`.

Example 16 Consider the summary graphs shown in Figure 18, which correspond to the summaries of Examples 14 and 15. The CP tables of edges (s_{10}, s_{11}) and (s_5, s_6) are the following:

Edge (s_{10}, s_{11}) CP table

parent	node	from	to	value
16	17	21	*Now*	"McGrady"
24	23	0	*Now*	"Garrity"
14	30	0	22	"Williams"

Edge (s_5, s_6) CP table

parent	node	from	to	value
6	7	0	20	"Oakley"
10	8	0	*Now*	–
16	17	0	20	"McGrady"
14	30	23	*Now*	"Williams"

Note that nodes 17 ("McGrady") and 30 ("Williams") appear in both tables but with different intervals. This happens because we are indexing cp's rather than nodes, and both nodes have two cp's ending at them.

For each temporal depth k, we will define a table called δ_k table, listing the nodes that are valid at certain intervals and their relative order. These intervals are obtained by taking all the intervals that label some continuous path of length k and partitioning them as needed to obtain a set of pairwise-disjoint intervals. This is formalized with the notion of *interval partition*.

Definition 20 (Interval Partition) The interval partition P of a set of intervals $I_1 \dots I_n$ is the smallest set of intervals $\mathcal{P} = P_1 \dots P_m$ such that all the P_i's in $\mathcal{P}$ are pairwise disjoint and $\mathcal{P}$ contains a partition of every interval I_j.

Definition 21 (δ_k Tables) Consider a temporal XML document D. For each temporal depth k in D there is a table called δ_k *table*. Each tuple `t` in a δ_k table has two temporal attributes, `from`, `to`, and a list-valued attribute `valid`. Let $I_1 \dots I_n$ be all the intervals such that there is a cp of length k labeled by one of the I_j's, and $P_1 \dots P_m$ be the interval partition of $I_1 \dots I_n$. Each P_k is represented by a tuple `t` in δ_k. The `t.valid` attribute contains the list of all nodes at temporal depth k that are valid in the interval `[t.from, t.to)`. The nodes in `t.valid` are ordered by the order relation defined in the interval `[t.from, t.to]`. (Note that, according to Proposition 1, this order relation is always defined for all nodes in `[t.from, t.to]`). Tuples in the δ_k tables are indexed by `from` and `to`.

Algorithm 5 (Construction of δ_k Tables) *The δ_k table construction algorithm starts by creating a temporary* event *table with two attributes,* `node` *and* `instant`*. For each tuple $\langle s_k, n, I\rangle$ in the tmap relation, where n (and s_k) are at depth k, two tuples* `t'` *and* `t''` *are created in the* event *table as follows:*

`t'.node` $= n$
`t'.instant` $= I.FROM$
`t''.node` $= n$
`t''.instant` $= I.TO$

The event *table is then sorted by the* `instant` *attribute (and the instant order $<_t$ when it is defined, i.e. when two or more tuples have the same* `instant` *attribute value t). Next the algorithm traverses the* event *table in ascending order adding and removing nodes from the* valid *node list. Nodes in the* valid *list are kept in the order defined in their intersection interval. Each entry in the* event *table represents a change in some node's state (from* valid *to* not valid *and viceversa). Therefore, for each tuple in the event table the algorithm checks whether the node is already in the* valid *list or not. If the node is in the list it means that the entry in the table corresponds to the end of its interval and therefore the node has to be removed from the* valid *list. If the node is not yet in the list, then the entry corresponds to the beginning of its interval and thus the node has to be added to the list. In addition, for each tuple in the* event *table the algorithm also checks if it is the last one in the table or if its instant attribute is different from the next. In both cases a tuple $\langle old - instant, instant - 1, valid\_list\rangle$ is added to the δ_k table, and no tuple is added otherwise.*

Example 17 We will apply the δ table construction algorithm to the s'_5 node of the interval summary in Figure 18. The *tmap* relation for s'_5 is the following:

tmap	
s'_5	$9, [0, Now]$
s'_5	$11, [0, Now]$
s'_5	$12, [23, Now]$
s'_5	$19, [0, 20]$

From the *tmap* relation for s'_5 the following *event* table is constructed:

δ_5 event table

node	instant
19	0
11	0
9	0
19	20
12	23
12	*Now*
11	*Now*
9	*Now*

Note that the order in which the nodes appear in the event table does not necessarily represent the instant order $<_t$. (For example, node '11' appears before node '19' in the temporal document at instant '0', rather than after it.) The instant order $<_t$ will be taken into account when the nodes are inserted in the *valid* list. Let us now begin to traverse the event table. The first node we find is '19' with instant '0'. We check if '19' is in the *valid* list. Since it is not (in fact the *valid* list is still empty at this point), we conclude that '0' corresponds to the beginning of '19's interval and we add it to the *valid* list. Likewise we add nodes '11' and '9'. Since the next tuple has a different instant value, we can now add the entry $\langle 0, \{9, 11, 19\}\rangle$ to the δ_5 table. Next we find node '19' with instant '20' and when we look it up in the *valid* list we find it. Thus we conclude that '20' corresponds to the end of '19's interval and hence we remove '19' from the list. We process the rest of the event table in a similar fashion and we get the following δ_5 table:

δ_5 table

from	to	valid
0	19	$\{9, 11, 19\}$
20	22	$\{9, 11\}$
23	*Now*	$\{9, 11, 12\}$

The δ_5 table contains the interval partition of intervals $[23, Now]$, $[0, 20]$, $[0, Now]$, which are the intervals of node s'_5 according to *tmap*.

The δ_k tables can be used for computing snapshots efficiently. When creating a snapshot at time i we simply have to find the tuple t in the δ_k tables such that i is contained in t's interval. In addition, the δ_k tables support efficient retrieval of all nodes that are valid during a given interval. In the next section we will explain query processing using the CP and δ_k tables in detail.

7.3 Space Requirements

The size of the index is proportional to the number of cp's. Our experiments in Section 10 show that, for the NBA database, the number of cp's is about three times the number of nodes in the temporal graph. We support three types of updates, insertion, deletion and modification. When the XML graph is a tree, *i.e.* before any update is performed, for each edge in the temporal graph there is one tuple in the CP tables. Furthermore, since there is only one interval of relevance, [0, Now], there is only one tuple t in each δ_k and the list of its valid nodes contains all nodes at temporal depth k. As updates are performed, the number of cp's in the document – and consequently the number of tuples in the tables – increases. The tables affected by an update are those that index descendants of a node at the update point, so the closer the update is to the root, the larger the increase in the index size. Occasionally, an update may also create a new partition in a δ_k table, in which case the nodes from the last partition that are still valid in the new partition have to be replicated.

There are several ways to reduce the space requirements for the index. In many applications, we expect most updates to occur close to the leaves, so that the size of the index will grow linearly in the size of the document. Our experiments so far confirm that expectation: the main-memory representation of TempIndex has a size comparable to that of the DOM representation (see Section 10) for all document sizes tested.

Another typical property of temporal applications is that there is a great deal of skew in the distribution of queries, with recent instants being accessed more frequently than older ones. In a space-constrained situation we could exploit this property by limiting how far the temporal window extends back in time, and periodically reindexing to take this into account.

8 Evaluating TXPath Queries Using TempIndex

In this section we will introduce the query evaluation algorithms which are based on our ancestor-descendant encoding.

8.1 Ancestor-Descendant Encoding for Temporal XML

So far we have used node numbers for identifying nodes in the XML graph. However, we will show that we can encode nodes in a more efficient way in order to improve the performance of some TXPath queries. We devised the *temporal interval encoding*, which is an ancestor-descendant encoding inspired by the interval scheme first presented by Santoro and Khatib [48]. In this scheme, the leaves of a tree are numbered from left to right and each internal node is labeled with a pair of numbers corresponding to its smallest and largest leaf descendants.

All known ancestor-descendant encoding schemes (see [30] for a recent survey) are variations of Santoro and Khatib's interval scheme. The average label length of these class of schemes has an upper bound of $2 \log n$, n being the number of nodes in the XML graph. In our index, the integration of the encoding with other index structures allows us to encode the ancestor-descendant relationship using only one number instead of two (the end of each interval is implicitly stored in the order of the δ_k tables).

The main idea for the *temporal interval encoding* is based on taking advantage of three facts: (a) again, we are indexing continuous paths, not just nodes; (b) the intervals of all the continuous paths in which a node n participates are disjoint; (c) the graph representing a snapshot of a temporal XML document is acyclic. Thus, we can encode the nodes in a way such that each node has as many encodings as continuous paths it is part of.

In order to formally define the temporal interval encoding, we need to define first a total order relation among nodes at different intervals.

Definition 22 Let $p_1 = (root, \ldots, v, T_1)$ and $p_2 = (root, \ldots, w, T_2)$ be continuous paths in D. The partial order relation $\prec_T$ is defined as follows:

1. If $T_1 \bigcap T_2 = \phi$ then $v \prec_{[0,Now]} w$ iff $T_1.FROM < T_2.FROM$.
2. If $T_1 \bigcap T_2 \neq \phi$ then $v \prec_T w$ iff $v <_t w$ for every $t \in T$, where $<_t$ is the order relation at instant t.

Definition 23 (Temporal Interval Encoding) Let $\prec_T$ be the order relation from Definition 22, and let $succ_{\prec}(n, T')$ be the successor function in $\prec_T$ of node n at interval $T' \subseteq T$ (a node may have different successors at different intervals). In addition, let $gap(n, T)$ be a function assigning an arbitrary integer to each node n in a given interval T (the *gap* function represents the "integer gap" between two consecutive encodings). The *temporal interval encoding* function τ is defined over pairs $\langle n, T\rangle$ such that there is a cp $p = (root, \ldots, n, T)$, as follows: $\tau(\langle n', T''\rangle) = \tau(\langle n, T'\rangle) + gap(n, T)$ where T is an interval such that $succ_{\prec}(n, T) = n'$, and $\tau(\langle n', T''\rangle) = 0$ otherwise.

The gap function is designed to specify how much "room" we want to leave between encodings for future

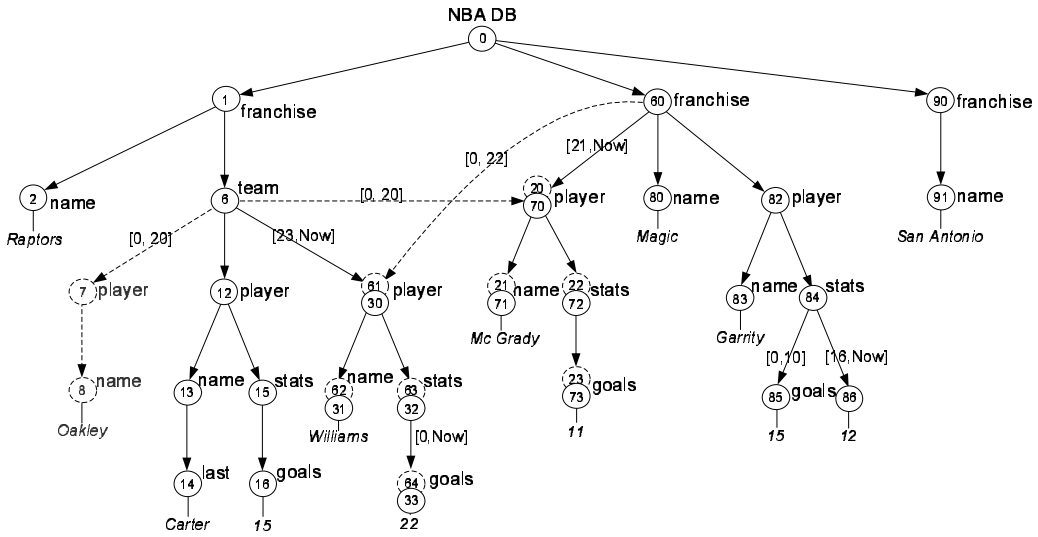

Fig. 19 Indexing intervals with *temporal interval encoding*

updates. For instance, in the temporal encoding of Figure 19, there is only one gap assignment to the node with encoding 6, which is the following: $gap(6, [0, 20]) = 1$ (the next encoding is 7 for the $[0, 20]$ interval). In contrast, the node 16 has three different gap assignments: $gap(16, [0, 20]) = 4$ (the next encoding for the $[0, 20]$ interval is 20), $gap(16, [21, 22]) = 44$ (the next encoding for the $[21, 22]$ interval is 60), and $gap(16, [23, Now]) = 14$ (the next encoding for the $[23, Now]$ interval is 30).

For representing the structural relationship between nodes in different equivalence classes using the temporal interval encoding we define what we call *TCP tables*. TCP tables are CP tables where nodes are represented by the temporal interval encoding. Since temporal encoding does not require explicitly representing nodes' parents, in the TCP tables the `parent` attribute is dropped. In contrast to CP tables, a TCP table is associated to a summary node rather than an edge.

Definition 24 (TCP Tables) Consider the summary S_D. For each node s in the temporal summary graph G_S there is a *TCP table* in which each tuple `t` has attributes `node`, `from` and `to` such that there is a continuous path from the root of d to `t.node` with interval `[t.from, t.to]`. When `t.node` has a value v associated to it, the TCP table has an extra attribute named `value`, where `t.value`$= v$. Tuples in the TCP tables are sorted by `node`.

Example 18 Consider for instance Figure 19. The *player* node corresponding to 'Williams' has initially been encoded as 61. This number encodes the node in the interval $[0, 22]$, when it was a descendant of node 60. For the interval $[23, Now]$, the node's number is 30, because it became a descendant of 1.

In other words, there are two continuous paths (with disjoint intervals) from the root to the node, and for each one of them we use a different encoding. Note that these different node numbers do not imply a larger number of tuples in the TCP tables with respect to the CP tables, because there is always one tuple for each continuous path, as in the encoding used before. For instance, consider the LCP summary graph shown in Figure 18 and the NBA example with the temporal encoding of Figure 19. The TCP tables of nodes s_{11} and s_6 are the following:

Node s_{11} TCP table

`node`	`from`	`to`	`value`
71	21	*Now*	"McGrady"
83	0	*Now*	"Garrity"
62	0	22	"Williams"

Node s_6 TCP table

`node`	`from`	`to`	`value`
8	0	20	"Oakley"
13	0	*Now*	–
21	0	20	"McGrady"
31	23	*Now*	"Williams"

These tables are the CP tables from Example 16 but with temporal encoding and without the `parent` attribute. Note that nodes with "McGrady" and "Williams" values appear in both tables but with different intervals and temporal encodings.

8.2 Query Evaluation

The evaluation of a TXPath query is divided into stages based on its *filter sections*. The filter sections of a TXPath query (also called *filters*) are the expressions that appear between brackets in the query. A filter is a predicate which is applied to the pairs $\langle node, interval \rangle$ that are at the end of the cp's that match the path expression before it. For simplicity, we consider in this section TXPath expressions without nested filters. After each filter

section, the evaluation of the rest of the query continues only for those pairs (node, interval) that satisfy the filter.

We decompose each TXPath query into a sequence of calls to six evaluation functions: `getParent(Label)`, `getDescendants(Label)`, `getChildren(Label)`, `getAncestors(Label)`, `valFilter(valPred)` and `tempFilter(TempPred)`, where `Label` is a node label, `valPred` is a value predicate, and `tempPred` is temporal predicate. Each function is evaluated on a list of *tmap* tuples $\langle s, n, I\rangle$ and returns another list of *tmap* tuples. In order to return a TXPath answer, the summary node s is dropped from the $\langle s, n, I\rangle$ tuples so that the list returned contains only pairs (n, I), just as the TXPath semantics requires.

Example 19 Consider the query "name of players who have played for the Toronto Raptors continuously since instant 20" which is expressed in TXPath as

```
//franchise[name='Raptors']//player/name[@from≥20]
```

This query can be evaluated top-down with the evaluation functions as follows:

```
list.add(root);
list = list.getDescendants(''franchise'');
list = list.getChildren(''name'');
list = list.valFilter('Raptors');
list = list.getParent(''franchise'');
list = list.getDescendants(''player'');
list = list.getChildren(''name'');
list = list.tempFilter(''from ≥ 20'');
```

If the number of nodes satisfying the last predicate ("from $\geq$ 20") is smaller than those satisfying the first predicate ('Raptors'), it might be better in terms of performance to choose a bottom-up query plan like the following:

```
list.add(leaves);
list1 = list.getAncestors(''name'');
list1 = list1.tempFilter(''from ≥ 20'');
list1 = list1.getParent(''player'');
list1 = list1.getAncestors(''franchise'');
list2 = list.getAncestors(''name'');
list2 = list2.valFilter('Raptors');
list2 = list2.getParent();
list = list1.intersect(list2);
list = list.getDescendants(''player'');
list = list.getChildren(''name'');
```

Query plans that are a blend of top-down and bottom-up evaluations are also possible.

We present next algorithms for computing functions `getDescendants(Label)` and `tempFilter(tempPred)` on a TempIndex using the temporal interval encoding.

Algorithm 6 (getDescendants)

INPUT: inList, Label.
OUTPUT: outList.

1. *For each s such that (s, n, I) is in inList*
   1.1. *Get the descendants of s in the summary graph with label Label and add them to sNodes.*
2. *For each s in sNodes*
   2.1. *For each n such that there is a tuple (s, n, I) in inList*
      2.1.1 *Get $(n', I') = succesor(n, I)$*
      2.1.2 *Assign to outList all tuples $(s, t.node, [t.from, t.to])$ such that t is a tuple in the TCP table of s, and $\tau(n, I) < t.node < \tau(n', I')$.*
3. *Return outList.*

Algorithm 7 (tempFilter)

INPUT: inList, tempPred.
OUTPUT: outList.

1. *For each n such that (s, n, I) is in inList.*
   1.1. *Get δ_x where x is the temporal depth of n during the interval I.*
   1.2. *For each interval i such that $(i, valid)$ is in δ_x and i satisfies tempPred*
      1.2.1. *Assign to outList all tuples (s, n, I) such that n is in valid and there is a tuple t in the TCP table of s such that $t.node = n$ and $I = (t.from, t.to)$.*
3. *Return outList.*

We illustrate next through and example how a TXPath query is evaluated using the evaluation functions and a TempIndex.

Example 20 Consider again the query of Example 19, expressed in TXPath as

```
//franchise[name='Raptors']//player/name[@from≥20]
```

We will follow the top-down evaluation presented in Example 19 on the LCP and interval summary graphs of Figure 18 and the NBA example with the temporal encoding of Figure 19. The evaluation begins by adding the tuple $\langle s_1, 0, [0, Now]\rangle$ to *list*, which contains the root elements of the XML document and the LCP summary graph. Then the evaluation continues by searching for the descendants of the summary root with label "franchise", which is s_2, and then its children with label "name", which is s_3. Since we have not filter out anything yet, *list* contains at this point all tuples in the TCP table of node s_3 (without the values):

Node s_3 TCP table

node	from	to	value
2	0	*Now*	"Raptors"
80	0	*Now*	"Magic"
91	0	*Now*	"San Antonio"

The next step is selecting the node that has the "Raptors" value (node 2), so that *list* is reduced now to the tuple $(2, [0, Now])$. The evaluation continues by going back to summary node s_2 in order to obtain the "franchise" node that corresponds to the name "Raptors". For this we will need the TCP table of s_2:

Node s_2 TCP table

node	from	to
1	0	*Now*
60	0	*Now*
90	0	*Now*

The parent node of 2 is the node with the biggest temporal encoding smaller than 2 in the s_2 TCP table, which is 1. Then, the tuple $\langle s_2, 1, [0, Now]\rangle$ is now assigned to *list* (the previous tuples are removed).

Since we have filtered out tuples from the TCP tables involved, we will need the entire encoding interval of node 1 to continue evaluating the descendants. The encoding interval will be used to determine exactly what nodes of all descendant TCP tables are in fact descendants of node 1. The right end of the interval is obtained from the $delta_1$ table

δ_1 table

from	to	valid
0	Now	$\{1, 60, 90\}$

by taking 60, the node next to 1 in the valid list of the appropriate interval (the only one in this case). The next step consist in obtaining all nodes with temporal encodings between 1 and 60 from the corresponding TCP tables. For that we first find the descendant "player" nodes in the LCP summary graph (s_5 and s_{10}), and then their "name" children (s_6 and s_{11}). The TCP tables of s_6 and s_{11} are the following:

Node s_{11} TCP table

node	from	to	value
71	21	Now	"McGrady"
83	0	Now	"Garrity"
62	0	22	"Williams"

Node s_6 TCP table

node	from	to	value
8	0	20	"Oakley"
13	0	Now	–
21	0	20	"McGrady"
31	23	Now	"Williams"

From these tables we select all nodes between 1 and 60 and add them to *list*, which now contains $\langle s_6, 8, [0, 20]\rangle$, $\langle s_6, 13, [0, Now]\rangle$, $\langle s_6, 21, [0, 20]\rangle$, $\langle s_6, 31, [23, Now]\rangle$. The last step consists of filtering *list* by selecting only those tuples `t` that have `t.FROM` ≥ 20, and this ends the evaluation.

9 Temporal Updates

In this section we describe the updates allowed over a temporal XML document. We will admit three kinds of changes over the document: insertion of a new node, deletion (in the sense of temporal databases) of a node, and update of containment edges. As usual, we will represent a labelled edge as a tuple $e(n_i, n_f, [t_i, t_f])$, where n_i and n_f are the initial and final nodes, and $[t_i, t_f]$ represents the interval of validity of the edge. Alternatively, as a shorthand, we will use T_e for denoting this interval. We will also describe how these updates are propagated to our temporal indexing scheme (assuming the *temporal interval encoding*). Finally, we will discuss how the concepts explained in Section 4 interplay with the updating process.

Since our model deals only with transaction time, all updates occur at the *current time instant*, denoted t_c. However, the update operators may be extended, allowing, if needed, some limited form of retroactive updating, without changing other characteristics of the model. We will discuss this issue below.

9.1 Insertion

The insertion of a new node in a temporal XML document requires specifying the new node n' to be inserted, and a current node n (i.e., a node with an incoming containment edge where $T_{e_c}.TO = Now$). The new node n', and a containment edge from n to n' with temporal label $[t_c, Now]$ are added to the graph. The DDL (Data Definition Language) syntax for insertion, along the lines of Tatarinov *et al.* [53], is:

```
FOR variable IN PathExpression
   INSERT ChildExpression
   [VALUE value]
```

`PathExpression` returns pairs $\langle node, interval\rangle$. For each pair such that $interval.TO = Now$, a new node is added as a child of *node*, with path label given by `ChildExpression`. If the new node is a value node, the `VALUE` keyword allows indicating the corresponding value. This keyword is omitted when inserting an element node.

Algorithm 8 (InsertNode)

INPUT: Document D, Summary S, insert_statement.
OUTPUT: Updated Document D and Summary S.

1. *Evaluate* `PathExpression` *in insert_statement with evaluation functions from Section 8.2.*
2. *For each current node v in the output of the previous step, let w' be the new child to be inserted with interval $[t_c, Now]$*
   - 2.1. *Get w, the last child of v at current time t_c which has a current interval $[t_w, Now]$. Let w'' a node in D such that $succ_{\prec}(w, T) = w''$ for some interval T.*
   - 2.2. *Insert w' into D and S by updating the order relation and the successor function as follows: $succ_{\prec}(w, T_1) = w'$, $succ_{\prec}(w', T_2) = w''$, and $succ_{\prec}(w, T_3) = w''$. (Note that intervals T_1, T_2, and T_3 are a partition of the former interval T and that $succ_{\prec}(w, T) = w''$ is no longer pertinent.)*
   - 2.3. *Update the gap function for w and w' as follows*
     - 2.3.1. *Set $gap(w, T_1) = gap(w, T) DIV\ 2$, where DIV is the integer division*
     - 2.3.2. *Set $gap(w', T_2) = gap(w, T) - gap(w, T_1)$*
     - 2.3.3. *Set $gap(w, T_3) = gap(w, T)$*
     - 2.3.4. *Delete $gap(w, T)$*
   - 2.4. *Since $succ_{\prec}(w, T_1) = w'$, then at this point the encoding of w' at interval $[t_c, Now]$ is assigned as follows: $\tau(\langle w', [t_c, Now]\rangle) = \tau(\langle w, [t_w, Now]\rangle) + gap(w, T_1)$*
3. *Update the corresponding TCP tables, creating a new table if required (i.e., if there is no table associated to the label that appears in* `ChildExpression`*).*
4. *Update the corresponding δ_i tables as follows*
   - 4.1. *Get l, the last tuple in δ_i.*

4.2. Set $l.to = t_c - 1$.
4.3. Add a new tuple r *s.t.* $r.from = t_c$, $r.to = Now$, $r.valid = l.valid \cup \{w''\}$.

Example 21 Let us consider the following expression evaluated on Figure 19 database:

```
FOR $p IN //player[name/last='Carter']/stats
   INSERT $p/minutes
   VALUE '33.2'
```

This update is processed as follows (assume for simplicity that t_c is the instant 120 and that the new node is w' with interval $[120, Now]$). We begin by evaluating the path expression in the first line, which returns the node with encoding 15, label *stats* and interval $[0, Now]$. (In this discussion, we will use encodings to identify nodes when possible.) Then, we locate the last (and only) child of 15 at current time, which is 16. At this point (before the insertion) the successor of 16 at interval $[23, Now]$ is 30. The next step is to insert w' by updating the order relation as follows: change the interval in which the successor of 16 is 30 from $[23, Now]$ to $[23, 119]$, set w' as successor of 16 during $[120, Now]$, and set 30 as successor of w' during $[120, Now]$.

The update continues with the gap function. We have that $gap(16, [23, Now]) = 14$ before the update. Then, we set $gap(16, [120, Now]) = 7$, $gap(16, [23, 119]) = 14$, and then delete $gap(16, [23, Now]) = 14$ because it is no longer valid. Next, we assign an encoding to w' with τ as follows: $\tau(\langle w', [120, Now]\rangle) = \tau(\langle 16, [0, Now]\rangle) + gap(16, [120, Now])$. Thus, $\tau(\langle w', [120, Now]\rangle) = 16 + 7 = 23$.

The final step is to update the TCP and δ_i tables. Since there is no node in the summary graph for w', we add a new node labeled *minutes* to the summary graph and create a new TCP table. Next, we insert the tuple $\langle 11, 120, Now, 33.2\rangle$ into the new TCP table. Finally, we update the δ_5 table by adding a tuple r with $r.from = 120$ at the end and the other attributes of r are set as follows. Let us denote l the last tuple in δ_5 immediately before the insertion of r. Thus, r becomes the last tuple in δ_5, with $r.from = 120$, $r.to = Now$, and $r.valid = l.valid \cup \{23\}$. Finally, we set $l.to = 119$.

We are assuming that updates are performed over a *consistent* document, and must leave this document in a *consistent* state. In the case of insertion of a node n, the new node has only one incoming edge, meaning that inconsistencies of type *ii* cannot occur. Also, the inserted node has no outgoing edges. Therefore, consistency conditions of types *i* (temporal label outside the lifespan of the node) and *iii* (cycles) cannot be introduced.

We commented above that the data model allows a limited form of retroactive update. We will briefly clarify this notion. Suppose we want to insert a new player node to the Orlando Magic franchise (i.e., an insertion below node 60 in Figure 19). Since the lifespan of node 60 is $[0, Now]$, instead of giving the new node the lifespan $[t_c, Now]$, we could specify any temporal label included in $[0, Now]$. We only need to add a statement to the operator's syntax, indicating the lifespan of the new node. Given that we only deal with transaction time, a complete discussion of this topic is beyond the scope of this paper.

9.2 Deletion

We can delete (in the temporal database sense) attribute nodes (except attributes of type ID), element nodes, and reference edges from a temporal XML document. Again, we only allow current objects to be deleted. Informally, when deleting a node n at time t_d, 'Now' is replaced by t_d in $T_{e_c}.TO$. The same occurs with all the containment edges in the current subtree of n (the subtree with root n where all the edges e_c have $T_{e_c}.TO = Now$). Reference edges are deleted by setting $T_{e_r}.TO = t_d$ in the temporal label of the edge. Notice that no consistency checking is required. Thus, this operation will always leave the document in a consistent state.

Like in the case of insertion discussed above, a retroactive deletion could be implemented if the consistency conditions of the model are satisfied. For example, in Figure 19 we could delete node 86 at any instant between 17 and Now.

Example 22 Suppose we want to 'delete' all statistics for Williams (node 30 in Figure 19). The DDL for this update will be:

```
FOR $p IN /NBAdb//player[name='Williams']
   DELETE node $p//stats
```

The deletion is processed as follows. Again, we assume that deletion can only occur at the present time.

We begin by processing the path expression in the first line using the summaries, returning node 30 along with its interval. The next step is the delete operation, which involves 'deleting' the subtree with root 32 at time '120'. In order to do that, we first replace the tuple $\langle 32, 23, Now\rangle$ in the TCP table of s_8 with $\langle 32, 23, 120\rangle$. Next, we replace $\langle 33, 23, Now\rangle$ in the TCP table of s_9 with $\langle 33, 23, 120\rangle$. Finally, we update the δ_4 and δ_5 tables by inserting a new tuple r (recall that the table is ordered according to the attribute *instant*), with $r.from = 120$, $r.to = Now$, and $l.to = 119$, where l is last tuple of each table. In addition, we set $r.valid = l.valid - \{32\}$, for δ_4 and $r.valid = l.valid - \{33\}$, for δ_5.

9.3 Edge updates

We will finish our discussion of temporal updates with updates of containment edges. Let D be a temporal XML document; n and n_i two current nodes in D such that

there exists a current containment edge from n_i to n. Let us consider another current node n_j, not in a current subtree of n; intuitively, a temporal update at instant t_c says that from t_c on, the parent of node n will be n_j.

Example 23 Suppose player "Garrity" starts playing for the Toronto Raptors at the present time (instant '120'):

```
FOR /NBAdb//player[name='Garrity']
   SET PARENT
        /NBAdb/franchise[name='Raptors']/team
```

We begin by processing the path expression in the first line using the summaries, returning node 82 along with its interval. As we are using temporal interval encoding, all the nodes in the subtree with root 82 (including node 82 itself) must be given a new node number. In this example, let us assume the following number assignments: 82 → 42, 83 → 43, 84 → 44, 85 → 45, 86 → 46. Next, we insert the tuple $\langle 42, 121, Now\rangle$ in the TCP table of s_5, and we replace the tuple $\langle 82, 0, Now\rangle$ with $\langle 82, 0, 120\rangle$ in the TCP table of s_{10}. In addition, , we insert the tuple $\langle 43, 121, Now, Garrity\rangle$ in the TCP table of s_6, and replace the tuple $\langle 83, 0, Now, Garrity\rangle$ with $\langle 83, 0, 120, Garrity\rangle$ in the TCP table of s_{11}. For the remaining elements, we perform an analogous procedure and update the affected δ_k tables.

Consistency checking gets a little more involved in this case. Besides verifying that the new parent node is current, we need to check consistency condition *iii*, i.e., that no cycle is introduced by the update (in the case of updates performed over current nodes, this limits to check that no cycles are introduced at the current instant.

10 Experiments

In this section we will show how indexing temporal intervals and continuous paths improves TXPath query evaluation. We compare TempIndex with other two systems: a traditional, non-temporal structural summary and a DOM-based structure. We have picked ToXin [46] as a representative of the the systems that are based on non-temporal path summaries. We choose this particular system for convenience, since it is easily available to us; but we believe the results would not be substantially different using any other structural summaries proposal discussed in Section 2. The second comparison will be against a DOM representation of the base data without any kind of summary.

Although using a non-temporal summary reduces the search space for TXPath queries – compared to the DOM approach – it does not help with the temporal semantics of the query evaluation. ToXin has data structures that summarizes specific fragments of the XML data, such as nodes, paths and subtrees, and thus avoid sequential scans of entire documents during query evaluation. However, like all traditional structural summaries, it materializes *paths* rather than *continuous paths*; therefore, both ToXin and DOM have to compute all continuous paths on-the-fly during query evaluation time. Our experiments show how important indexing the temporal structure of the data base is for evaluating TXPath queries.

Doc (MB)	# of XML data nodes	# of *cp's*	# of TempIndex summary nodes
20	540300	1694010	94
40	1080600	3388020	94
60	1620900	5082030	94
80	2161200	6776040	94
100	2701500	8470050	94

Fig. 20 Benchmark data sets and index parameters

Doc (MB)	TempIndex (MB)	Toxin (MB)	DOM (MB)
20	95	201	165
40	190	402	330
60	285	604	495
80	380	805	660
100	475	906	825

Fig. 21 Main-memory data structures sizes

Query	TXPath template
Q1	`//Player/Name`
Q2	`//APG`
Q3	`//Div[Name='X']/Player[Interval='I']`
Q4	`//SEQUENCE[APG≥'n' and Interval='I']` `/ancestor::Player/Name`
Q5	`for $p in //Player[Name='X']` `INSERT newNode $p//APG VALUE 'V'`
Q6	`for $p in //Player[Name='X']` `DELETE node $p//stats`
SN	`Snapshot`

Fig. 22 Benchmark TXPath query templates

TempIndex is implemented in Java 2 and uses Berkeley DB Java Edition [50] as persistent storage. This is a substantial difference with respect to the implementation presented in [37], which was a pure main-memory system. In the current TempIndex implementation all data structures are loaded into main-memory during query evaluation and update processing, and are saved to disk afterwards. This allows us to run queries on databases much larger than the main-memory available by loading and saving different index fragments during evaluation time. We have also optimized the internal representation of time intervals and attributes, with the consequent reduction in the index size with respect to [37].

For all our experiments we use query processing time as the performance metric. We evaluate the performance of the three systems on a set of seven query templates, as shown in Figure 22. Query templates that contain value and interval selections (Q3 through Q6) were tested with

Query	Doc. size				
	20 MB	40 MB	60 MB	80 MB	100 MB
Q1	2006	4012	6018	8024	10030
Q2	16776	33552	50328	67104	83880
Q3	504	1008	1512	2016	2520
Q4	1174	2348	3522	4696	5870
Q5	104	208	312	416	520
Q6	8	16	24	32	40

Fig. 23 Answer sizes of retrieval queries (Q1 to Q4) and number of update points of update queries (Q5 and Q6)

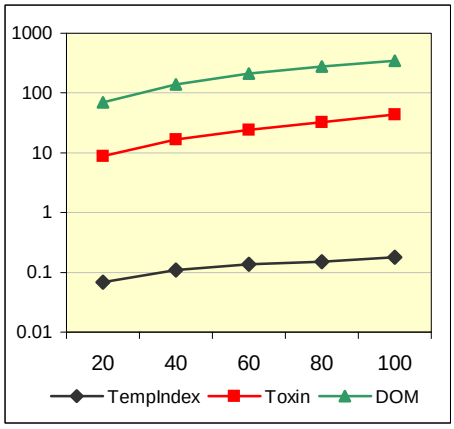

Fig. 24 Query Q1 – log scale

ten different actual queries with various combinations of values and intervals. For those four queries we report the average results. The results for the SN template are also an average of computing ten document snapshots at ten different instants.

Templates Q1 and Q2 are TXPath retrieval queries without interval constraints. Even though the intervals are not specified in the expression, these are still TXPath queries and thus the answers are pairs $\langle node, interval \rangle$. Templates Q3 and Q4 have interval constraints. Half of the queries tested for Q3 and Q4 were *snapshot queries*, i.e., queries where the interval 'I' was actually a time instant. Remember that in Section 6 we distinguished snapshot queries from the document reconstruction as of a given time instant, which we denoted *document snapshot*(reported in the SN template). A document snapshot represents the state of the database at a given point in time. In other words, it is a query of the form "Give the state of the NBA database as of October 10, 1995". Therefore, while the answer to queries Q1 to Q4 are $\langle node, interval \rangle$ pairs, the answer to a document snapshot is an XML document. Finally, Q5 and Q6 are TXPath update queries, as described in Section 9.

We run our benchmark queries over the NBA database, which we consider to be a representative example of temporal data. We loaded the data from the NBA web site (www.nba.com) into a relational database (Microsoft SQL Server 2000.) From this database we produced five

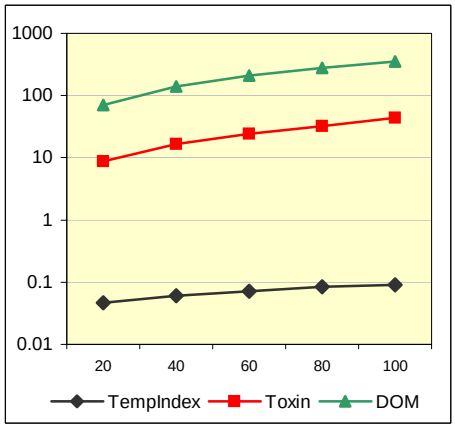

Fig. 25 Query Q2 – log scale

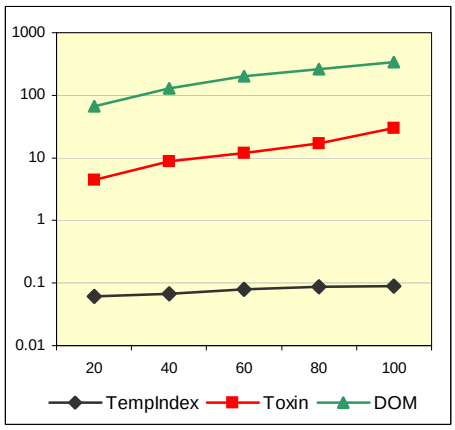

Fig. 26 Query Q3 – log scale

documents of 20, 40, 60, 80, and 100 Megabytes. We ran all queries over the five documents and the results are reported in Figures 24 to 30. For the experiments we used a Pentium 4 PC at 2Ghz with 1GB of RAM memory and a 60 GB hard drive. We report the number of nodes and continuous paths in the temporal documents, as well as the number of summary nodes in TempIndex in Figure 20. The size of the query answer for each query is shown in Figure 23.

In all retrieval queries TempIndex performed faster than ToXin. The TempIndex speed-up against ToXin ranged from a minimum of nine times (document snapshot–20MB) to a maximum of 220 times (Q2–100MB). Since both systems summarize label paths and values, the difference in performance can be mostly attributed to the summarization of continuous paths.

Q2 is one of the fastest in TempIndex but one of the slowest in ToXin. The reason for that is that the answer to Q2 is a whole class of continuous paths in the temporal index, which is very easy to find and retrieve using the TempIndex summary graph. Although in ToXin we can

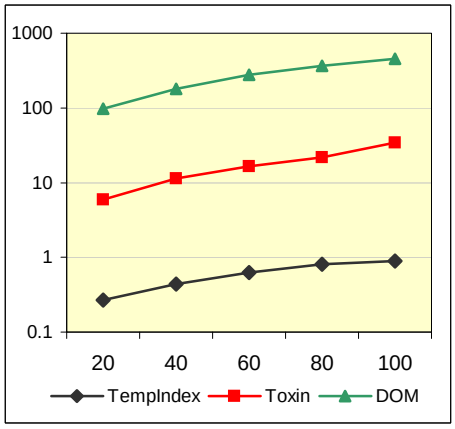

Fig. 27 Query Q4 – log scale

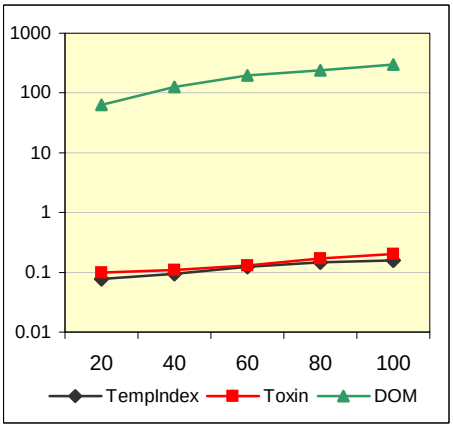

Fig. 29 Query Q6: Delete – log scale

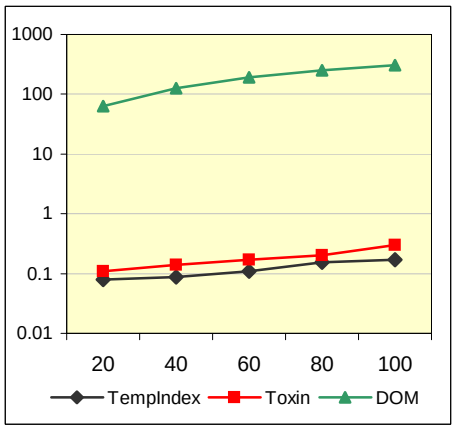

Fig. 28 Query Q5: Insert – log scale

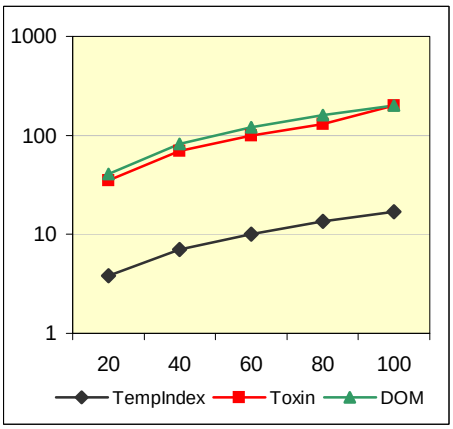

Fig. 30 Snapshot – log scale

narrow the search by following only those label paths that match the regular expression in the query, we still have to compute all continuous paths over them.

The document snapshots, in contrast, require heavy computation even for TempIndex. We can still narrow the search considerably by using the interval index to locate the classes corresponding to the instant in time we are looking for. However, once these classes are found we have to reconstruct an entire document navigating back and forth over them. That being said, TempIndex still is almost one order of magnitude faster than ToXin and DOM. Since a non-temporal path summary is not very efficient for temporal document reconstruction, the snapshot computation performance of ToXin and DOM are quite similar.

Queries Q1 and Q2 do not contain either interval or value selection predicates and have relatively large answer sets. However, keep in mind that not having interval predicates does not mean that the temporal semantics is not present: the continuous paths always have to be computed in order to return TXPath query answers. This is the reason behind the two orders of magnitude difference in performance between ToXin and TempIndex for queries Q1 and Q2.

The answer sets of Q1 are closer to the root and smaller than those of Q2. This affects the query processing time in ToXin because the continuous paths to be computed are fewer and much shorter in Q1 than in Q2, with the consequent impact on query evaluation (Q2 queries take almost twice the time than Q1 ones). In contrast, since the DOM implementation is not aware of the label path structure of the data graph, it requires the traversal of the whole temporal graph in order to match the regular expression on both Q1 and Q2. Consequently, the difference in query processing time between Q1 and Q2 is minimal in DOM.

Queries Q3 and Q4 require the additional computation of value and interval selection, which is reflected in the TempIndex results. In contrast, the size of the answer set and the length of the continuous paths seems to have a bigger impact on ToXin performance than the selection operations, and almost no impact at all in DOM.

The reason for that seems to be that ToXin spends most of the query processing time on continuous path computations, while DOM does it on data graph traversal.

Update queries Q5 (insert) and Q6 (delete) require label path traversal in order to locate the update point. Since no continuous path computation is involved, the difference between ToXin and TempIndex is minimal. In contrast, the DOM implementation has to traverse the whole temporal graph in order to locate the update point, with the consequent time difference against both ToXin and TempIndex.

11 Conclusion

In this paper we studied the problem of modeling and querying temporal data in XML. We first proposed an abstract data model for temporal XML, and compare this model against other proposals, pointing out benefits and limitations. We discussed four different alternatives for implementing the abstract data model as temporal XML documents. Based on our data model we studied the problem of validating temporal XML documents against the temporal constraints that the data model imposes. This problem has been overlooked in other proposals of temporal data models for XML. We gave algorithms for checking the presence of temporal inconsistencies in a document and fixing them, and studied the algorithms' complexity.

We also studied the problem of indexing temporal XML documents. For this, we first introduced a temporal XML query language denoted TXPath, that extends the semantics of XPath 2.0 to return sequences of (node, interval) pairs instead of just sequences of nodes. The indexing scheme we proposed is based on the materialization of *continuous paths* instead of paths. A new class of summaries, denoted *TSummaries*, that adds the time dimension to the usual path summarization schemes, serves as framework to our indexing scheme. We presented two new kinds of summaries: *LCP* and *Interval* summaries. The indexing scheme, denoted TempIndex, integrates these summaries, also including other data structures.

We compared the performance of a persistent implementation of TempIndex, against a traditional non-temporal structural summary (ToXin), and a DOM-based structure. This comparison highlights the benefits of materializing *continuous paths*. TempIndex ran one order of magnitude faster than ToXin and DOM, for snapshots. For retrieval queries, TempIndex ran, on the average, from 10 to 210 times faster than the other schemes. In addition, we sketched a language for updates, and showed that the cost of updating the index is compatible with real-world requirements.

Future work includes extending the study of new classes of Temporal Summaries, for their application to different settings. We also believe that our work on consistency issues can be a good starting point for studying and reasoning about constraints with indeterminate dates, of the types presented in [16,28]. The problem of reasoning about temporal constraints in XML is still underexplored.

Acknowledgements. The work presented in this paper is the continuation of a research project started jointly with our beloved friend and mentor Alberto O. Mendelzon, who sadly passed away in June, 2004.

We are grateful to the reviewers for their hard work and invaluable insights which helped to greatly improve the paper. We would also like to thank Mariana Zerega, who collaborated in the implementation of many of the algorithms presented in this work, and Marcela Campo, for her help with the algorithms presented in Section 4.

Alejandro Vaisman was partially supported by the Millennium Nucleus Center for Web Research, Grant P04-67-F, Mideplan, Chile.

References

1. Serge Abiteboul, Sophie Cluet, Guy Ferran, and Marie-Christine Rousset. The Xyleme project. *Computer Networks 39(3)*, pages 225–238, 2002.
2. T. Amagasa, M. Yoshikawa, and S. Uemura. A temporal data model for XML documents. In *Proceedings of DEXA Conference*, pages 334–344, 2000.
3. T. Bozkaya and M. Ozsoyoglu. Indexing valid time intervals. In *Proceedings of DEXA Conference*, pages 541–550, 1998.
4. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. *Computer Networks 39(5)*, pages 473–487, 2002.
5. P. Buneman, S. Khanna, K Tajima, and W. Tan. Archiving scientific data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 1–12, Madison, USA, 2002.
6. S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. In *Theory and Practice of Object Systems, Vol 5(3)*, pages 143–162, 1999.
7. S. Chawathe, H. G.Molina, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of 100th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, 1994.
8. S. Chien, V. Tsotras, and C. Zaniolo. Version management of XML documents. In *Proceedings of the Third International Workshop on the Web and Databases*, pages 75–80, Dallas, TX, 2000.
9. S. Chien, V. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 291–300, Rome, Italy, 2001.
10. J. Chomicki. Temporal query languages: a survey. In *Proceedings of the 1st International Conference on Temporal Logic,LNAI 827*, pages 506–534, 1994.
11. Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An adaptive path index for XML data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 121–132, 2002.
12. J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the semantics of "now" in databases. *ACM Trans. Database Syst.*, 22(2):171–214, 1997.

13. Mariano P. Consens and Tova Milo. Optimizing queries on files. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 301–312, 1994.
14. S. De Capitani. An authorization model for temporal XML documents. In *Proceedings of SAC'02*, pages 1088–1093, Madrid, Spain, 2002.
15. Natasha Drukh, Neoklis Polyzotis, Minos N. Garofalakis, and Yossi Matias. Fractional XSKETCH synopses for XML databases. In *Second International XML Database Symposium, XSym 2004*, pages 189–203, 2004.
16. C. Dyreson and R. Snodgrass. Supporting valid-time indeterminacy. *ACM Transactions on Database Systems*, 23(1):1–57, 1998.
17. C.E. Dyreson. Observing transaction-time semantics with TTXPath. In *Proceedings of WISE 2001*, pages 193–202, 2001.
18. C.E. Dyreson, M.H. Bolen, and C.S. Jensen. Capturing and querying multiple aspects of semistructured data. In *Proceedings of the 25th VLDB Conference*, pages 290–301, 1999.
19. O. Etzion, S. Jajodia, and S. Sripada (eds.). *Temporal Databases: Research and Practice*. Springer-Verlag, LNCS 1399, 1998.
20. W. Fan and Jérôme Siméon. Integrity constraints for XML. *Journal of Computer and Systems Sciences, 66(1)*, pages 254–291, 2003.
21. D. Florescu and D. Kossmann. Storing and querying XML data using a RDBMS. *IEEE Data Engineering Bulletin, 22(3)*, pages 27–34, 1999.
22. C. Gao and R. Snodgrass. Syntax, semantics and query evaluation in the τXQuery temporal XML query language. *Time Center Technical Report TR-72*, 2003.
23. C. Gao and R. Snodgrass. Temporal slicing in the evaluation of XML queries. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 632–643, Berlin, Germany, 2003.
24. M. Gergatsoulis and Y. Stavrakas. Representing changes in XML documents using dimensions. In *Proceedings of the First Symposium on XML databases (XSym 2003)*, pages 208–222, Berlin, Germany, 2003.
25. Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445, 1997.
26. F. Grandi. Introducing an annotated bibliography on temporal and evolution aspects in the world wide web. *SIGMOD Record 33(2)*, pages 4–86, 2004.
27. F Grandi and F. Mandreoli. The valid web: an XML/XSL infrastructure for temporal management of web documents. In *Proceedings of the International Conference on Advances in Information Systems*, pages 294–303, 2000.
28. F. Grandi and F. Mandreoli. Effective representation and efficient management of indeterminate dates. In *TIME'01*, pages 164–169, 2001.
29. Hao He and Jun Yang. Multiresolution indexing of XML for frequent queries. In *Proceedings of the 20th International Conference on Data Engineering*, pages 683–694, 2004.
30. H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the thirteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 954–963, 2002.
31. Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 133–144, 2002.
32. Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Pradeep Shenoy. Updates for structure indexes. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 239–250, 2002.
33. Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 129–140, 2002.
34. Hartmut Liefke and Dan Suciu. XMILL: An efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.
35. M.G. Manukyan and L.A. Kalinichenko. Temporal XML. In *Proceedings of ADBIS*, pages 581–590, Vilnius, Lithuania, 2001.
36. A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *Proceedings of the 27th VLDB Conference*, pages 581–590, Rome, Italy, 2001.
37. Alberto O. Mendelzon, Flavio Rizzolo, and Alejandro Vaisman. Indexing temporal XML documents. In *Proceedings of the 30th International Conference on Very Large Databases*, pages 216–227, Toronto, Canada, 2004.
38. Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory*, pages 277–295, 1999.
39. Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *Proceedings of the 13th International Conference on Data Engineering*, pages 79–90, 1997.
40. B Oliboni, E. Quintarelli, and L. Tanca. Temporal aspects of semistructured data. *Proceedings of the Eight International Symposium of Temporal Representation and Reasoning*, pages 119–127, 2001.
41. Neoklis Polyzotis and Minos N. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 358–369, 2002.
42. Neoklis Polyzotis and Minos N. Garofalakis. Structure and value synopses for XML data graphs. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 466–477, 2002.
43. Neoklis Polyzotis and Minos N. Garofalakis. XCLUSTER synopses for structured XML content. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006.
44. Neoklis Polyzotis, Minos N. Garofalakis, and Yannis E. Ioannidis. Approximate XML query answers. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 263–274, 2004.
45. Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 134–144, 2003.
46. Flavio Rizzolo and Alberto O. Mendelzon. Indexing XML data with ToXin. In *Proceedings of 4th International Workshop on the Web and Databases*, pages 49–54, 2001.
47. B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys, vol. 31,no. 2,pp 158-221*, 1999.
48. N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal (28)*, pages 5–8, 1985.
49. Ralf Schenkel, Anja Theobald, and Gerhard Weikum. HOPI: An efficient connection index for complex XML document collections. In *Proceedings of the 9th Conference on Extending Database Technology*, pages 237–255, 2004.
50. Sleepycat Software. *Berkeley DB Java Edition*, 2006. http://www.sleepycat.com/products/bdbje.html.

51. Richard Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.
52. A. Tansel, J. Clifford, and S. Gadia (eds.). *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings, 1993.
53. I. Tatarinov, G. Ives, A. Halevy, and D. Weld. Updating XML. In *Proceedings of ACM SIGMOD Conference*, pages 413–424, Santa Barbara, California, 2001.
54. P. Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies*, pages 183–202, Philadelphia, 1999.
55. F. Wang and C. Zaniolo. Temporal queries in XML document archives and web warehouses. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning (TIME'03)*, pages 47–55, Cairns, Australia, 2003.
56. F. Wang and C. Zaniolo. XBiT: An XML-based bitemporal data model. In *Proceedings of the 23rd International Conference on Conceptual Modeling*, pages 810–824, Shanghai, China, 2004.
57. F. Wang, X. Zhou, and C. Zaniolo. Efficient XML-based techniques for archiving, querying and publishing the histories of relational databases. In *Time Center TeEchnical Report*, 2005.
58. F. Wang, X. Zhou, and C. Zaniolo. Temporal XML? SQL strikes back! In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 47–55, Burlington, USA, 2005.
59. World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, 2002. http://www.w3.org/TR/2002/WD-xquery-20021115.
60. World Wide Web Consortium. *XML Path Language XPath 2.0*, 2003. http://www.w3.org/TR/2003/WD-xpath20-20030502.
61. Ke Yi, Hao He, Ioana Stanoi, and Jun Yang. Incremental maintenance of XML structural indexes. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 491–502, 2004.