# Supporting temporal slicing in XML databases*

Federica Mandreoli[1], Riccardo Martoglia[1], and Enrico Ronchetti[1]

DII, Università degli Studi di Modena e Reggio Emilia,
via Vignolese, 905/b - I 41100 Modena
⟨fmandreoli, rmartoglia, eronchetti⟩@unimo.it

**Abstract.** Nowadays XML is universally accepted as the standard for structural data representation; XML databases, providing structural querying support, are thus becoming more and more popular. However, XML data changes over time and the task of providing efficient support to queries which also involve temporal aspects goes through the tricky task of time-slicing the input data. In this paper we take up the challenge of providing a native and efficient solution in constructing an XML query processor supporting temporal slicing, thus dealing with non-conventional application requirements while continuing to guarantee good performance in traditional scenarios. Our contributions include a novel temporal indexing scheme relying on relational approaches and a technology supporting the time-slice operator.

## 1 Introduction

Nowadays XML is universally accepted as the standard for structural data representation and exchange and its well-known peculiarities make it a good choice for an ever growing number of applications. Currently the problem of supporting structural querying in XML databases is thus an appealing research topic for the database community.

As data changes over time, the possibility to deal with historical information is essential to many computer applications, such as accounting, banking, law, medical records and customer relationship management. In the last years, researchers have tried to provide answers to this need by proposing models and languages for representing and querying the temporal aspect of XML data. Recent works on this topic include [5, 9, 10, 12].

The central issue of supporting most temporal queries in any language is time-slicing the input data while retaining period timestamping. A time-varying XML document records a version history and temporal slicing makes the different states of the document available to the application needs. While a great deal of work has been done on temporal slicing in the database field [8], the paper [9] has the merit of having been the first to raise the temporal slicing issue in the XML context, where it is complicated by the fact that timestamps are distributed throughout XML documents. The solution proposed in [9] relies on a

* This work has been supported by the MIUR-PRIN Project: "European Citizen in eGovernance: legal-philosophical, legal, computer science and economical aspects".

stratum approach whose advantage is that they can exploit existing techniques in the underlying XML query engine, such as query optimization and query evaluation. However, standard XML query engines are not aware of the temporal semantics and thus it makes more difficult to map temporal XML queries into efficient "vanilla" queries and to apply query optimization and indexing techniques particularly suited for temporal XML documents.

In this paper we propose a native solution to the temporal slicing problem. In other words, we address the question of how to construct an XML query processor supporting time-slicing. The underlying idea is to propose the changes that a "conventional" XML pattern matching engine would need to be able to slice time-varying XML documents. The advantage of this solution is that we can benefit from the XML pattern matching techniques present in the literature, where the focus is on the structural aspects which are intrinsic also in temporal XML data, and that, at the same time, we can freely extend them to become temporally aware. Our ultimate goal is not to design a temporal XML query processor from scratch but to put at the user disposal an XML query processor which is able to support non-conventional application requirements while continuing to guarantee good performance in traditional scenarios.

We begin by providing some background in Section 2, where the temporal slicing problem is defined. Our main contributions are:

- We propose a novel temporal indexing scheme (Section 3.1), which adopts the inverted list technology proposed in [14] for XML databases and changes it in order to allow the storing of time-varying XML documents. Moreover, we show how a time-varying XML document can be encoded in it.
- We devise a flexible technology supporting temporal slicing (Section 3.2 to Section 3.5). It consists in alternative solutions supporting temporal slicing on the above storing scheme, all relying on the holistic twig join approach [2], which is one of the most popular approaches for XML pattern matching. The proposed solutions act at the different levels of the holistic twig join architectures with the aim of limiting main memory space requirements, I/O and CPU costs. They include the introduction of novel algorithms and the exploitation of different access methods.
- Finally, in Section 4 we present experimental results showing the substantial performance benefits achievable by combining the proposed solutions in different querying settings.

We describe related work and concluding remarks in Section 5.

## 2 Preliminaries: Notation and Temporal Slicing Definition

A time-varying XML document records a version history, which consists of the information in each version, along with timestamps indicating the lifetime of that version [5]. The left part of Fig. 1 shows the tree representation of our reference time-varying XML document taken from a legislative repository of norms. Data

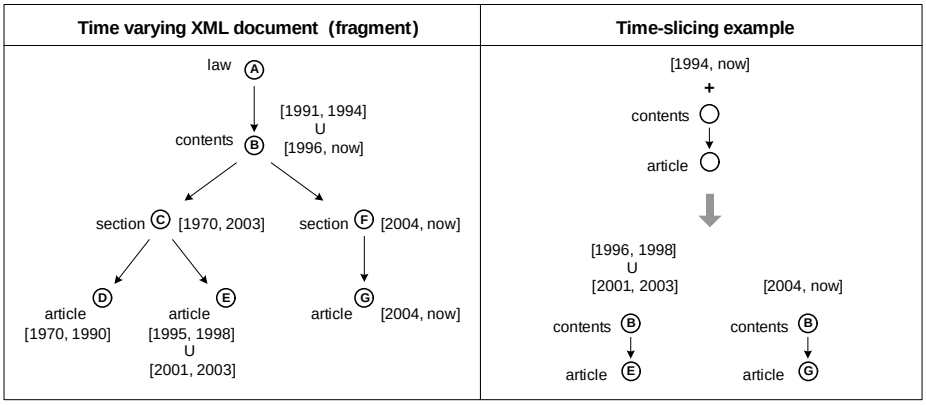



**Fig. 1.** Reference example.

nodes are identified by capital letters. For simplicity's sake, timestamps are defined on a single time dimension and the granularity is the year. Temporal slicing is essentially the snapshot of the time-varying XML document(s) at a given time point but, in its broader meaning, it consists in computing simultaneously the portion of each state of time-varying XML document(s) which is contained in a given period and which matches with a given XML query twig pattern. Moreover, it is often required to combine the results back into a period-stamped representation [9] in the period $[1994, now]$ and for the query twig `contents//article`. The right part of Fig. 1 shows the output of a temporal slicing example. This section introduces a notation for time-varying XML documents and a formal definition for the temporal slicing problem.

### 2.1 Document representation

A temporal XML model is required when there is the need of managing temporal information in XML documents and the adopted solution usually depends on the peculiarities of the application one wants to support. For the sake of generality, our proposal is not bound to a specific temporal XML model. On the contrary, it is able to deal with time-varying XML documents containing timestamps defined on an arbitrary number of temporal dimensions and represented as temporal elements [8], i.e. disjoint union of periods, as well as single periods.

In the following, we will refer to time-varying XML documents by adopting part of the notation introduced in [5]. A *time-varying XML database* is a collection of XML documents, also containing time-varying documents. We denote with $D^T$ a *time-varying XML document* represented as an ordered labelled tree containing timestamped elements and attributes (in the following denoted as nodes) related by some structural relationships (ancestor-descendant, parent-child, preceding-following). The timestamp is a temporal element chosen from one or more temporal dimensions and records the lifetime of a node. Not all

nodes are necessarily timestamped. We will use the notation $n^T$ to signify that node $n$ has been timestamped and $lifetime(n^T)$ to denote its lifetime. Sometimes it can be necessary to extend the lifetime of a node $n^{[T]}$, which can be either temporal or snapshot, to a temporal dimension not specified in its timestamp. In this case, we follow the semantics given in [6]: If no temporal semantics is provided, for each newly added temporal dimension we set the value on this dimension to the whole time-line, i.e. $[t_0, t_\infty)$.

The *snapshot* operator is an auxiliary operation which extracts a complete snapshot or *state* of a time-varying document at a given instant and which is particularly useful in our context. Timestamps are not represented in the snapshot. A snapshot at time $t$ replaces each timestamped node $n^T$ with its non-timestamped copy $x$ if $t$ is in $lifetime(n^T)$ or with the empty string, otherwise. The snapshot operator is defined as $snp(t, D^T) = D$ where $D$ is the snapshot at time $t$ of $D^T$.

### 2.2 The time-slice operator

The time-slice operator is applied to a time-varying XML database and is defined as `time-slice(twig,t-window)`. The `twig` parameter is a non-temporal node-labeled twig pattern which is defined on the snapshot schema [5] of the database through any XML query languages, e.g. XQuery, by specifying a pattern of selection predicates on multiple elements having some specified tree structured relationships. It defines the portion of interest in each state of the documents contained in the database. It can also be the whole document. The `t-window` parameter is the temporal window on which the time-slice operator has to be applied. More precisely, by default temporal slicing is applied to the whole time-lines, that is by using every single time point contained in the time-varying documents. With `t-window`, it is possible to restrict the set of time points by specifying a collection of periods chosen from one or more temporal dimensions.

Given a twig pattern `twig`, a temporal window `t-window` and a time-varying XML database *TXMLdb*, a *slice* is a mapping from nodes in `twig` to nodes in *TXMLdb*, such that: (i) query node predicates are satisfied by the corresponding document nodes thus determining the tuple $(n_1^{[T]}, \ldots, n_k^{[T]})$ of the database nodes that identify a distinct match of `twig` in *TXMLdb*, (ii) $(n_1^{[T]}, \ldots, n_k^{[T]})$ is *structurally consistent*, i.e. the parent-child and ancestor-descendant relationships between query nodes are satisfied by the corresponding document nodes, (iii) $(n_1^{[T]}, \ldots, n_k^{[T]})$ is *temporally consistent*, i.e. its lifetime $lifetime(n_1^{[T]}, \ldots, n_k^{[T]}) = lifetime(n_1^{[T]}) \cap \ldots \cap lifetime(n_k^{[T]})$ is not empty and it is contained in the temporal window, $lifetime(n_1^{[T]}, \ldots, n_k^{[T]}) \subseteq$ `t-window`. For instance, in the reference example, the tuple (B,D) is structurally but not temporally consistent as $lifetime(\text{B}) \cap lifetime(\text{D}) = \emptyset$. In this paper, we consider the temporal slicing problem:

> Given a twig pattern `twig`, a temporal window `t-window` and a time-varying XML database *TXMLdb*, for each distinct slice $(n_1^{[T]}, \ldots, n_k^{[T]})$,

`time-slice(twig,t-window)` computes the snapshot $snp(t, (n_1^{[T]}, \ldots, n_k^{[T]}))$, where $t \in lifespan(n_1^{[T]}, \ldots, n_k^{[T]})$.

Obviously, it is possible to provide a period-timestamped representation of the results by associating each distinct state $snp(t, (n_1^{[T]}, \ldots, n_k^{[T]}))$ with its pertinence $lifetime(n_1^{[T]}, \ldots, n_k^{[T]})$ in `t-window`.

## 3 Providing a native support for temporal slicing

In this paper we propose a native solution to the temporal slicing problem. To this end, we addressed two problems: The indexing of time-varying XML databases and the definition of a technology for XML query processing relying on the above indexing scheme and efficiently implementing the time-slice operator.

Existing work on "conventional" XML query processing (see, for example, [14]) shows that capturing the XML document structure using traditional indices is a good solution, on which it is possible to devise efficient structural or containment join algorithms for twig pattern matching. Being timestamps distributed throughout the structure of XML documents, we decided to start from one of the most popular approaches for XML query processing whose efficiency in solving structural constraints is proved. In particular, our solution for temporal slicing support consists in an extension to the indexing scheme described in [14] such that time-varying XML databases can be implemented and in alternative changes to the holistic twig join technology [2] in order to efficiently support the time-slice operator in different scenarios.

### 3.1 The temporal indexing scheme

The indexing scheme described in [14] is an extension of the classic inverted index data structure in information retrieval which maps elements and strings to inverted lists. The position of a string occurrence in the XML database is represented in each inverted list as a tuple `(DocId, LeftPos,LevelNum)` and, analogously, the position of an element occurrence as a tuple `(DocId, LeftPos:RightPos,LevelNum)` where (a) `DocId` is the identifier of the document, (b) `LeftPos` and `RightPos` can be generated by counting word numbers from the beginning of the document `DocId` until the start and end of the element, respectively, and (c) `LevelNum` is the depth of the node in the document. In this context, structural relationships between tree nodes can be easily determined: (i) *ancestor-descendant*: A tree node $n_2$ encoded as $(D_2, L_2 : R_2, N_2)$ is a descendent of the tree node $n_1$ encoded as $(D_1, L_1 : R_1, N_1)$ iff $D_1 = D_2$, $L_1 < L_2$, and $R_2 < R_1$; (ii) *parent-child*: $n_2$ is a child of $n_1$ iff it is a descendant of $n_1$ and $L_2 = L_1 + 1$.

As temporal XML documents are XML documents containing time-varying data, they can be indexed using the interval-based scheme described above and thus by indexing timestamps as "standard" tuples. On the other hand, timestamped nodes have a specific semantics which should be exploited when documents are accessed and, in particular, when the time-slice operation is applied.

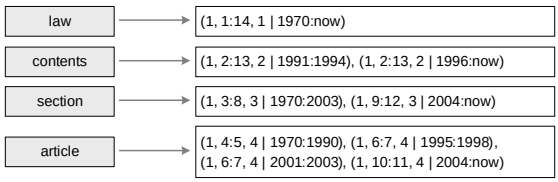



**Fig. 2.** The temporal inverted indices for the reference example

Our proposal adds time to the interval-based indexing scheme by substituting the inverted indices in [14] with *temporal inverted indices*. In each temporal inverted index, besides the position of an element occurrence in the time-varying XML database, the tuple `(DocId, LeftPos:RightPos,LevelNum|TempPer)` contains an implicit temporal attribute [8], `TempPer`. It consists of a sequence of `From:To` temporal attributes, one for each involved temporal dimension, and represents a period. Thus, our temporal inverted indices are in 1NF and each timestamped node $n^T$, whose lifetime is a temporal element containing a number of periods, is encoded through as many tuples having the same projection on the non-temporal attributes `(DocId, LeftPos:RightPos,LevelNum)` but with different `TempPer` values, each representing a period. All the temporal inverted indices are defined on the same temporal dimensions such that tuples coming from different inverted indices are always comparable from a temporal point of view. Therefore, given the number $h$ of the different temporal dimensions represented in the time-varying XML database, `TempPer` is $\texttt{From}_1\texttt{:To}_1\texttt{,...,From}_h\texttt{:To}_h$.

In this context, each time-varying XML document to be inserted in the database undergoes a pre-processing phase where (i) the lifetime of each node is derived from the timestamps associated with it, (ii) in case, the resulting lifetime is extended to the temporal dimensions on which it has not been defined by following the approach described in Subsec. 2.1. Fig. 2 illustrates the structure of the four indices for the reference example. Notice that the snapshot node A, whose label is `law`, is extended to the temporal dimension by setting the pertinence of the corresponding tuple to $[1970, now]$.

### 3.2 A technology for the time-slice operator

The basic four level architecture of the holistic twig join approach is depicted in Fig. 3. The approach maintains in main-memory a chain of linked stacks to compactly represent partial results to root-to-leaf query paths, which are then composed to obtain matches for the twig pattern (level SOL in Figure). In particular, given a path involving the nodes $q_1, \ldots, q_n$, the two stack-based algorithms presented in [2], one for path matching and the other for twig matching, work on the inverted indices $I_{q_1}, \ldots, I_{q_n}$ (level L0 in Figure) and build solutions from the stacks $S_{q_1}, \ldots, S_{q_n}$ (level L2 in Figure). During the computation, thanks to a deletion policy the set of stacks contains data nodes which are guaranteed to lie on a root-to-leaf path in the XML database and thus represents in linear

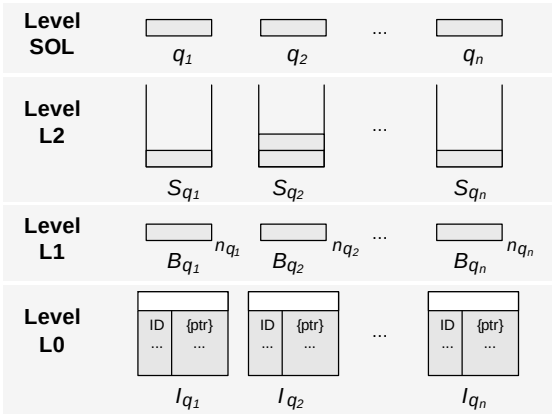



**Fig. 3.** The basic holistic twig join four level architecture

```
While there are nodes to be processed
(1) Choose the next node n_q̄
(2) Apply the deletion policy
(3) Push the node n_q̄ into the pertinence stack S_q̄
(4) Output solutions
```

**Fig. 4.** Skeleton of the holistic twig join algorithms (HTJ algorithms)

space a compact encoding of partial and total answers to the query twig pattern. The skeleton of the two holistic twig join algorithms (HTJ algorithms in the following) is presented in Fig. 4. At each iteration the algorithms identify the next node to be processed. To this end, for each query node $q$, at level L1 is the node in the inverted index $I_q$ with the smaller `LeftPos` value and not yet processed. Among those, the algorithms choose the node with the smaller value, let it be $n_{\bar{q}}$. Then, given knowledge of such node, they remove partial answers form the stacks that cannot be extended to total answers and push the node $n_{\bar{q}}$ into the stack $S_{\bar{q}}$. Whenever a node associated with a leaf node of the query path is pushed on a stack, the set of stacks contains an encoding of total answers and the algorithms output these answers. The algorithms presented in [2] have been further improved in [3, 11]. As our solutions do not modify the core of such algorithms, we refer interested readers to the above cited papers.

The time-slice operator can be implemented by applying minimal changes to the holistic twig join architecture. The time-varying XML database is recorded in the temporal inverted indices which substitute the "conventional" inverted index at the lower level of the architecture and thus the nodes in the stacks will be represented both by the position and the temporal attributes. Given a twig pattern `twig`, a temporal window `t-window`, a slice is the snapshot of any answer to `twig` which is temporally consistent. Thus the holistic twig join algorithms continue to work as they are responsible for the structural consistency of the slices and pro-

vide the best management of the stacks from this point of view. Temporal consistency, instead, must be checked on each answer output of the overall process. In particular, for each potential slice $((D, L_1 : R_1, N_1|T_1), \ldots, (D, L_k : R_k, N_k|T_k))$ it is necessary to intersect the periods represented by the values $T_1, \ldots, T_k$ and then check both that such intersection is not empty and that it is contained in the temporal window. Finally, the snapshot operation is simply a projection of the temporally consistent answers on the non-temporal attributes. In this way, we have described the "first step" towards the realization of a temporal XML query processor. On the other hand, the performances of this first solution are strictly related to the peculiarities of the underlying database. Indeed, XML documents usually contain millions of nodes and this is absolutely true in the temporal context where documents record the history of the applied changes. Thus, the holistic twig join algorithms can produce a lot of answers which are structurally consistent but which are eventually discarded as they are not temporally consistent. This situation implies useless computations due to an uncontrolled growth of the the number of tuples put on the stacks.

Temporal consistency considers two aspects: The intersection of the involved lifetimes must be non-empty (*non-empty intersection constraint* in the following) and it must be contained in the temporal window (*containment constraint* in the following). We devised alternative solutions which rely on the two different aspects of temporal consistency and act at the different levels of the architecture with the aim of limiting the number of temporally useless nodes the algorithms put in the stacks. The reference architecture is slightly different from the one presented in Fig. 3. Indeed, in our context, any timestamped node whose lifetime is a temporal element is encoded into more tuples (e.g. see the encoding of the timestamped node E in the reference example). Thus, at level L1, each node $n_q$ must be interpreted as the set of tuples encoding $n_q$. They are stored in buffer $B_q$ and step 3 of the HTJ algorithms empties $B_q$ and pushes the tuples in the stack $S_q$.

### 3.3 Non-empty intersection constraint

Not all temporal tuples which enter level L1 will at the end belong to the set of slices. In particular, some of them will be discarded due to the non-empty intersection constraint. The following Lemma characterizes this aspect. Without lose of generality, it only considers paths as the twig matching algorithm relies on the path matching one.

**Proposition 1.** *Let $(D, L : R, N|T)$ be a tuple belonging to the temporal inverted index $I_q$, $I_{q_1}, \ldots, I_{q_k}$ the inverted indices of the ancestors of $q$ and $TP_{q_i} = \bigcup \sigma_{\texttt{LeftPos}<L}(I_{q_i})|_{\texttt{TempPer}}$, for $i \in [1, k]$, the union of the temporal pertinences of all the tuples in $I_{q_i}$ having `LeftPos` smaller than $L$. Then $(D, L : R, N|T)$ will belong to no slice if the intersection of its temporal pertinence with $TP_{q_1}, \ldots, TP_{q_k}$ is empty, i.e. $T \cap TP_{q_1} \cap \ldots \cap TP_{q_k} = \emptyset$.*

Notice that, at each step of the process, the tuples having `LeftPos` smaller than $L$ can be in the stacks, in the buffers or still have to be read from the inverted

**Input:** Twig pattern `twig`, the last processed node $n_{\overleftarrow{q}}$
**Output:** Next node $n_{\bar{q}}$ to be processed
**Algorithm** `Load`:

```
(1) if all buffers are empty
(2)   start=root(twig);
(3) else
(4)   start=←q ;
(5) for each query node q from start to leaf(twig)
(6)    get n_q;
(7)    min_q is the minimum between n_q.LeftPos and min_parent(q);
(8)    if n_q.LeftPos is equal to min_q
(9)      load n_q into B_q;
(10)return the last node inserted into the buffers
```

**Fig. 5.** The buffer loading algorithm `Load`

indices. However, looking for such tuples in the three levels of the architecture would be quite computationally expensive. Thus, in the following we introduce a new approach for buffer loading which allows us to look only at the stack level. Moreover, we avoid accessing the temporal pertinence of the tuples contained in the stacks by associating a temporal pertinence to each stack (*temporal stack*). Such a temporal pertinence must therefore be updated at each push and pop operation. At each step of the process, for efficiency purposes both in the update and in the intersection phase, such a temporal pertinence is the smaller multi-dimensional period $P_q$ containing the union of the temporal pertinence of the tuples in the stack $S_q$.

The aim of our buffer loading approach is to avoid loading the temporal tuples encoding a node $n^{[T]}$ in the pertinence buffer $B_q$ if the inverted indices associated with the parents of $q$ contain tuples with `LeftPos` smaller than that of $n_q$ and not yet processed. Such an approach is consistent with step 1 of the HTJ algorithms as it chooses the node at level L1 with the smaller `LeftPos` value and ensures that when $n^{[T]}$ enters $B_q$ all the tuples involved in Prop. 1 are in the stacks. The algorithm implementing step 1 of the HTJ algorithms is shown in Fig. 5. We associate each buffer $B_q$ with the minimum $min_q$ among the `LeftPos` values of the tuples contained in the buffer itself and those of its ancestors. Assuming that all buffers are empty, the algorithm starts from the root of the `twig` (step 2) and, for each node $q$ up to the leaf, it updates the minimum $min_q$ and inserts $n_q$, the node in $I_q$ with the smaller `LeftPos` value and not yet processed, if it is smaller than $min_q$. The same applies when some buffers are not empty. In this case, it starts from the query node matching with the previously processed data node and it can be easily shown that the buffers of the ancestors of such node are not empty whereas the buffers of the subpath rooted by such node are all empty.

**Lemma 1.** *Assume that step 1 of the HTJ algorithms depicted in Fig. 4 is implemented by the algorithm* `Load`*. The tuple* $(D, L : R, N|T)$ *in* $B_q$ *will belong*

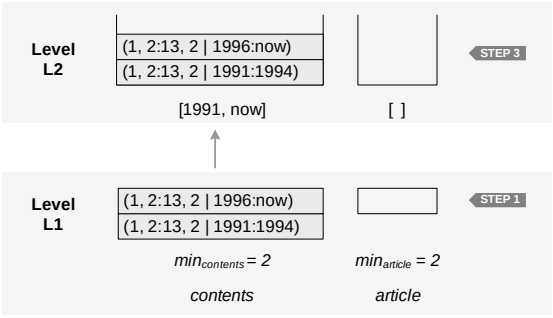



**Fig. 6.** State of levels L1 and L2 during the first iteration

*to no slice if the intersection of its temporal pertinence $T$ with the multidimensional period $P_{q_1 \to q_k} = P_{q_1} \cap \ldots \cap P_{q_k}$ intersecting the periods of the stacks of the ancestors $q_1, \ldots, q_k$ of $q$ is empty.*

For instance, at the first iteration of the HTJ algorithms applied to the reference example, step 1 and step 3 produce the situation depicted in Fig. 6. Notice that when the tuple $(1, 4 : 5, 4|1970 : 1990)$ encoding node D (label `article`) enters level L1 all the tuples with `LeftPos` smaller than 4 are already at level L2 and due to the above Lemma we can state that it will belong to no slice.

Thus, the non-empty intersection constraint can be exploited to prevent the insertion of useless nodes into the stacks by acting at level L1 and L2 of the architecture. At level L2 we act at step 3 of the HTJ algorithms by simply avoiding pushing into the stack $S_q$ each temporal tuple $(D, L : R, N|T)$ encoding the next node to be processed which satisfies Lemma 1, i.e. such that $T \cap P_{q_1 \to q_k} = \emptyset$. At level L1, instead, we act at step 9 of the algorithm `Load` by avoiding loading in any buffer $B_q$ each temporal tuple encoding $n_q$ which satisfies Lemma 1. More precisely, given the `LeftPos` value of the last processed node, say $CurLeftPos$, we only load each tuple $(D, L : R, N|T)$ such that $L$ is the minimum value greater than $CurLeftPos$ and $T$ intersects $P_{q_1 \to q_k}$. To this purpose, our solution uses time-key indices combining the `LeftPos` attribute with the attributes $\texttt{From}_j\texttt{:To}_j$ in the `TempPer` implicit attribute representing one temporal dimension in order to improve the performances of range-interval selection queries on the temporal inverted indices. In particular, we considered two access methods: The B+-tree and a temporal index, the Multiversion B-tree (MVBT) [1].

An one-dimensional index like the B+-tree, clusters data primarily on a single attribute. Thus, we built B+-trees that cluster first on the `LeftPos` attribute and than on the interval end time $\texttt{To}_j$. In this way, we can take advantage of sequential I/O as tree leaf pages are linked and records in them are ordered. In particular, we start with the first leaf page that contains a `LeftPos` value greater than $CurLeftPos$ and a $\texttt{To}_j$ value greater than or equal to $P_{q_1 \to q_k}|_{\texttt{From}_j}$, i.e. the projection of the period $P_{q_1 \to q_k}$ on the interval start time $\texttt{From}_j$. Then we proceed by loading the records until the leaf page with the next `LeftPos` value

or with a $\texttt{From}_j$ value greater than $P_{q_1 \to q_k}|_{\texttt{To}_j}$ is met. This has the effect of selecting each tuple $(D, L : R, N|T)$ where $L$ is the smaller value greater than $CurLeftPos$ and its period $T|_{\texttt{From}_j:\texttt{To}_j}$ intersect the period $P_{q_1 \to q_k}|_{\texttt{From}_j:\texttt{To}_j}$, as $T|_{\texttt{To}_j} \geq P_{q_1 \to q_k}|_{\texttt{From}_j}$ and $T|_{\texttt{From}_j} \leq P_{q_1 \to q_k}|_{\texttt{To}_j}$.

The alternative approach we considered is to maintain multiple versions of a standard B+-tree through an MVBT. An MVBT index record contains a key, a time interval and a pointer to a page and, thus, this structure is able to directly support our range-interval selection requirements.

### 3.4 Containment constraint

The following proposition is the equivalent of Prop. 1 when the containment constraint is considered.

**Proposition 2.** *Let $(D, L : R, N|T)$ be a tuple belonging to the temporal inverted index $I_q$. Then $(D, L : R, N|T)$ will belong to no slice if the intersection of its temporal pertinence with the temporal window* `t-window` *is empty.*

It allows us to act at level L1 and L2, but also between level L0 and level L1. At level L1 and L2 the approach is the same as the non-empty intersection constraint; it is sufficient to use the temporal window `t-window`, and thus Prop. 2, instead of Lemma 1. Moreover, it is also possible to add an intermediate level between level L0 and level L1 of the architecture, which we call "under L1" (UL1), where the only tuples satisfying Prop. 2 are selected from each temporal inverted index, are ordered on the basis of their `(DocId,LeftPos)` values and then pushed into the buffers. Similarly to the approach explained in the previous section, to speed up the selection, we exploit $B^+$-tree indices built on one temporal dimension. Notice that this solution deals with buffers as streams of tuples and thus it provides interesting efficiency improvements only when the temporal window is quite selective.

### 3.5 Combining solutions

The non-empty intersection constraint and the containment constraint are orthogonal thus, in principle, the solutions presented in the above subsections can be freely combined in order to decrease the number of useless tuples we put in the stacks. Each combination gives rise to a different scenario denoted as "X/Y", where "X" and "Y" are the employed solutions for the non-empty intersection constraint and for the containment constraint, respectively (e.g. scenario L1/L2 employs solution L1 for the non-empty intersection constraint and solution L2 for the containment constraint). Some of these scenarios will be discussed in the following. First, scenario L1/UL1 is not applicable since in solution UL1 selected data is kept and read directly from buffers, with no chance of additional indexing. Instead, in scenario L1/L1 the management of the two constraints can be easily combined by querying the indices with the intersection of the temporal pertinence of the ancestors (Proposition 1) and the required temporal window.

All other combinations are straightforwardly achievable, but not necessarily advisable. In particular, when L1 is involved for any of the two constraints the L1 indices have to be built and queried: Therefore, it is best to combine the management of the two constraints as in L1/L1 discussed above. Finally, notice that the baseline scenario is the SOL/SOL one, involving none of the solutions discussed in this paper.

## 4 Experimental Evaluation

In this section we present the results of an actual implementation of our XML query processor supporting temporal slicing showing its behavior on different document collections and in different execution scenarios.

### 4.1 Experimental setting

The document collections follow the structure of the documents used in [10], where three temporal dimensions are involved, and have been generated by a configurable XML generator. On average, each document contains 30-40 nodes, a depth level of 10, 10-15 of these nodes are timestamped nodes $n^T$, each one in 2-3 versions composed by the union of 1-2 distinct periods. We are also able to change the length of the periods and the probability that the temporal pertinence of the document nodes overlap. Finally, we investigate different kinds of probability density functions generating collections with different distributions, thus directly affecting the containment constraint.

Experiments were conducted on a reference collection (C-R), consisting of 5000 documents (120 MB) generated following a uniform distribution and characterized by not much scattered nodes, and on several variations of it. We tested the performance of the time-slice operator with different `twig` and `t-window` parameters. Due to the lack of space, in this article we will deepen the performance analysis by considering the same path, involving three nodes, and different temporal windows as our focus is not on the structural aspects.

The experiments have been performed on a Pentium 4 3Ghz Windows XP Professional workstation, equipped with 1GB RAM and an 160GB EIDE HD with NT file system (NTFS).

### 4.2 Efficiency evaluation

We evaluated the performances of the time-slice operator in terms of execution time and number of tuples that are put in the buffers and in the stacks for each feasible computation scenario.

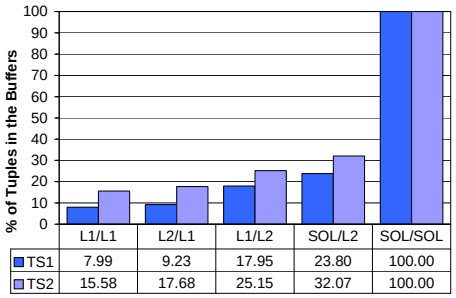
**Evaluation of the default setting.** We started by testing the time-slice operator with a default setting (denoted as TS1 in the following). Its temporal window has a selectivity of 20%, i.e. 20% of the tuples stored in the temporal inverted indexes involved by the twig pattern intersect the temporal window. The

| Evaluation scenarios: | Execution Time (ms) | Non-Consistent Solutions (%) | Tuples (%) Buffer | Stack |
|---|---|---|---|---|
| **L1/L1** | 1890 | 23.10 % | 7.99 % | 7.76 % |
| **L2/L1** | 1953 | 23.10 % | 9.23 % | 7.76 % |
| **SOL/L1** | 2000 | 39.13 % | 9.43 % | 9.17 % |
| **L1/L2** | 2625 | 23.10 % | 17.95 % | 7.76 % |
| **L2/L2** | 2797 | 23.10 % | 23.37 % | 7.76 % |
| **SOL/L2** | 2835 | 39.13 % | 23.80 % | 9.17 % |
| **L1/SOL** | 12125 | 95.74 % | 88.92 % | 88.85 % |
| **L2/SOL** | 12334 | 95.74 % | 99.33 % | 88.85 % |
| **SOL/SOL** | 12688 | 96.51 % | 100.00 % | 100.00 % |

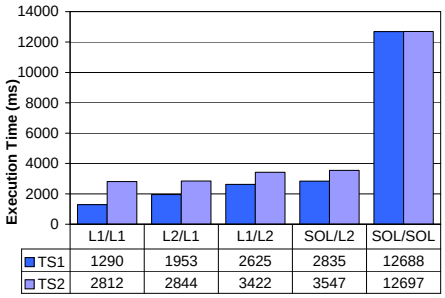
**Table 1.** Evaluation of the computation scenarios with TS1.

returned solutions are 5584. Table 1 shows the performance of each scenario when executing TS1. In particular, from the left: The execution time, the percentage of potential solutions at level SOL that are not temporally consistent and, in the last two columns, the percentage of tuples that are put in the buffers and in the stacks w.r.t. the total number of tuples involved in the evaluation. Notice that, the temporal inverted indices exploited at level L1 are B+-trees; the comparison of the performances between the $B^+$-tree and MVBT implementations will be shown in the following.

The best result is given by the computation scenario L1/L1: Its execution time is more than 6 times faster than the execution time of the baseline scenario SOL/SOL. Such a result clearly shows that combining solutions at a low level of the architecture, such as L1, avoids I/O costs for reading unnecessary tuples and their further elaboration cost at the upper levels. The decrease of read tuples from 100% of SOL/SOL to just 7.99% of L1/L1 and the decrease of temporally inconsistent solutions from 96.51% of SOL/SOL to 23.1% of L1/L1 represent a remarkable result in terms of efficiency. Let us now have a look to the other scenarios. TS1 represents a typical querying setting where the containment constraint is much more selective than the non-empty intersection constraint. This consideration induces us to analyse the obtained performances by partitioning the scenarios in three groups, */L1, */L2 and */SOL, on the basis of the adopted containment constraint solution. The scenarios within each group show similar execution time and percentages of tuples. In group */L1 the low percentage of tuples in buffers (10%) means low I/O costs and this has a good influence on the execution time. In group */L2 the percentages of tuples in buffers are more than double of those of group */L1, while the execution time is about 1.5 times higher. Finally, group */SOL is characterized by percentages of tuples in buffers and execution time approximately ten and six time higher than those in */T1, respectively. Moreover, within each group it should be noticed that rising the non-empty intersection constraint solution from level L1 to level SOL produces more and more deterioration in the overall performances.

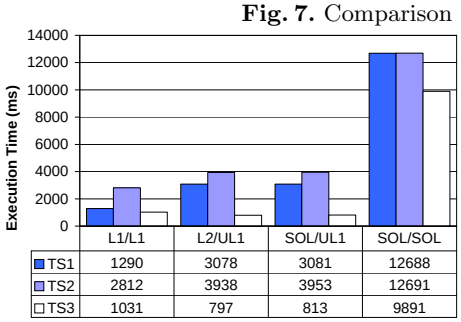



(a) Percentage of tuples in the buffers

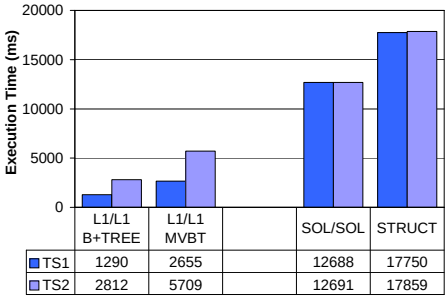



(b) Execution Time (ms)

**Fig. 7.** Comparison between TS1 and TS2.




(a) UL1 scenarios performances

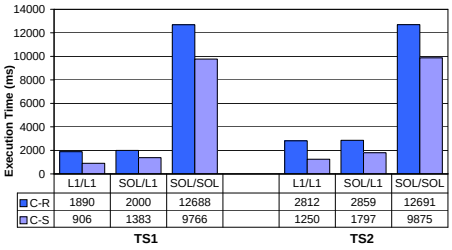



(b) MVBT and structural approach performances

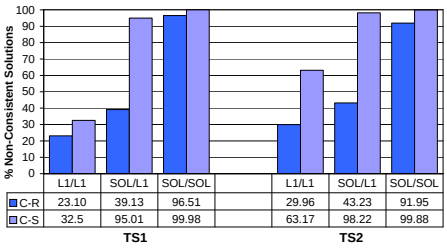
**Fig. 8.** Additional execution time comparisons.

**Changing the selectivity of the temporal window.** We are now interested in showing how our XML query processor responds to the execution of temporal slicing with different selectivity levels; to this purpose we considered a second time-slice (TS2) having a selectivity of 31% (lower than TS1) and returning 12873 solutions. Figure 7 shows the percentage of read tuples (Figure 7-a) and the execution time (Figure 7-b) of TS1 compared with our reference time-slice setting (TS1). Notice that the trend of growth of the percentage of read tuples along the different scenarios is similar. However, for TS1 the execution time follows the same trend as the read tuples whereas for TS2 the execution time of different scenarios are closer. In this case, the lower selectivity of the temporal window makes the benefits achievable by the L1 solutions less appreciable. Notice that, in the SOL/SOL scenario both queries have the same number of tuples in the buffers because no selectivity is applied at the lower levels; this explains also the same execution time.

**Evaluation of the performance of solution UL1.** In order to evaluate the results of exploiting access methods at level UL1 we considered a third time-slice (TS3) that is characterized by a highly selective temporal window (1%)




(a) Execution time




(b) Percentage of Non-Consistent Solutions

**Fig. 9.** Comparison between the two collections C-R and C-S.

and returns 123 solutions. Figure 8-a compares the execution time of the scenarios involving UL1 solutions (*/UL1) with the best and the baseline scenarios shown above (L1/L1 and SOL/SOL). As one would expect, it shows that */UL1 scenarios are inefficient for low-selectivity settings, while they are the best ones with high-selectivity setting. In particular the best computation scenario for TS3 is L2/UL1.

**Comparison with MVBT and purely structural techniques.** In Figure 8-b we compare the execution time for scenario L1/L1 when the access method is the B+-tree w.r.t. the MVBT. Notice that when MVBT indices are used to access data the execution time is generally higher than the B+-tree solution. This might be due to the implementation we used which is a beta-version included in the XXL package [7]. The last comparison involves the holistic twig join algorithms applied on the original indexing scheme proposed in [14] where temporal attributes are added to the index structure but are considered as common attributes. Notice that in this indexing scheme tuples must have different `LeftPos` and `RightPos` values and thus each temporal XML document must be converted into an XML document where each timestamped node gives rise to a number of distinct nodes equal to the number of distinct periods. The results are shown on the right of Figure 8-b where it is clear that the execution time of the purely structural approach (STRUCT) is generally higher than our baseline scenario and thus also than the other scenarios (13 times slower than the best scenario). This demonstrates that the introduction of our temporal indexing scheme alone brings significant benefits on temporal slicing performance. We refer the interested reader also to Section 5 where we provide additional discussion of state of the art techniques w.r.t. ours.

**Evaluation on differently distributed collections.** We also considered the performance of our XML query processor on another collection (C-S) of the same size of the reference one, but that is characterized by temporally scattered nodes. Figure 9 shows the execution time and the number of temporally inconsistent

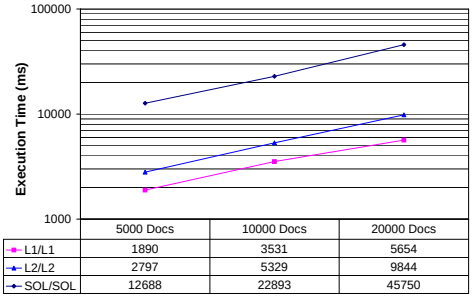



**Fig. 10.** Scalability results for TS1.

potential solutions of TS1 and TS2 on both collections. The execution time of scenarios L1/L1 and SOL/L1, depicted in Figure 9-a, shows that it is almost unchanged for collection C-R, whereas the difference is more remarkable for both temporal slicing settings for collection C-S. Notice also that the percentage of temporally inconsistent potential solutions when no solution is applied under level SOL is limited in the C-R case but explodes in the C-S case (see for instance SOL/L1 in Fig. 9-b). The non-empty intersection constraint is mainly influenced by the temporal sparsity of the nodes in the collection: The more the nodes are temporally scattered the more the number of temporally inconsistent potential solutions increases. Therefore, when temporal slicing is applied to this kind of collections the best way to process it is to adopt a solution exploiting the non-empty intersection constraint at the lowest level, i.e. L1.

**Scalability.** Figure 10 (notice the logarithmic scales) reports the performance of our XML query processor in executing TS1 for the reference collection C-R and for two collections having the same characteristics but different sizes: 10000 and 20000 documents. The execution time grew linearly in every scenario, with a proportion of approximately 0.75 w.r.t. the number of documents for our best scenario L1/L1. Such tests have also been performed on the other temporal slicing settings where we measured a similar trend, thus showing the good scalability of the processor in every type of query context.

## 5 Discussion and Concluding Remarks

In the last years, there has been a growing interest in representing and querying the temporal aspect of XML data. Recent papers on this topic include those of Currim et al. [5], Gao and Snodgrass [9], Mendelzon et al. [12], and Grandi et al. [10] where the history of changes XML data undergo is represented into a single document from which versions can be extracted when needed. In [5], the authors study the problem of consistently deriving a scheme for managing the temporal counterpart of non-temporal XML documents, starting from the definition of their schema. The paper [9] presents a temporal XML query language, $\tau$XQuery, with which the authors add temporal support to XQuery by extending

its syntax and semantics to three kinds of temporal queries: Current, sequenced, and representational. Similarly, the TXPath query language described in [12] extends XPath for supporting temporal queries. Finally, the main objective of the work presented in [10] has been the development of a computer system for the temporal management of multiversion norms represented as XML documents and made available on the Web.

Closer to our definition of `time-slice` operator, Gao and Snodgrass [9] need to time-slice documents in a given period and to evaluate a query in each time slice of the documents. The authors suggest an implementation based on a stratum approach to exploit the availability of XQuery implementations. Even if they propose different optimizations of the initial time-slicing approach, this solution results in long XQuery programs also for simple temporal queries and postprocessing phases in order to coalesce the query results. Moreover, an XQuery engine is not aware of the temporal semantics and thus it makes more difficult to apply query optimization and indexing techniques particularly suited for temporal XML documents. Native solutions are, instead, proposed in [4, 12]. The paper [4] introduces techniques for storing and querying multiversion XML documents. Each time one or more updates occur on a multiversion XML document, the proposed versioning scheme creates a new physical version of the document where it stores the differences w.r.t. the previous version. This leads to large overheads when "conventional" queries involving structural constraints and spanning over multiple versions are submitted to the system. In [12] the authors propose an approach for evaluating TXPath queries which integrates the temporal dimension into a path indexing scheme by taking into account the available continuous paths from the root to the elements, i.e. paths that are valid continuously during a certain time interval. While twig querying is not directly handled in this approach, path query performance is enhanced w.r.t. standard path indexing, even though the main memory representation of their indices is more than 10 times the size of the original documents. Moreover, query processing can still be quite heavy for large documents, as it requires the full navigation of the document collection structure, in order to access the required element tables, and the execution of a binary join between them at each level of the query path.

Similarly to the structural join approach [14] proposed for XML query pattern matching, the temporal slicing problem can be naturally decomposed into a set of temporal-structural constraints. For instance solving `time-slice(//contents//section//article,`$[1994, now]$`)` means to find all occurrences in a temporal XML database of the basic ancestor-descendant relationships `(contents,section)` and `(section,article)` which are temporally consistent. In the literature, a great deal of work has been devoted to the processing of temporal join (see e.g. [13]) also using indices [15]. Given the temporal indexing scheme proposed in this paper, we could have extended temporal join algorithms to the structural join problem or vice versa. However the main drawback of the structural join approach is that the sizes of the results of binary structural joins can get very large, even when the input and the final result sizes obtained by stitching together the basic matches are much more manageable.

The native approach proposed in this paper extends one of the most efficient approaches for XML query processing and the underlying indexing scheme in order to support temporal slicing and overcome most of the previously discussed problems. Starting from the holistic twig join approach [2], which directly avoids the problem of very large intermediate results size by using a chain of linked stacks to compactly represent partial results, we proposed new flexible technologies consisting in alternative solutions and extensively experimented them in different settings. The resulting good efficiency is quite encouraging and induces us to continue in this direction.

## References


1. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4), 1996.
2. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. of the ACM SIGMOD*, pages 310–321, 2002.
3. T. Chen, J. Lu, and T. Wang Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *Proc. of the ACM SIGMOD*, 2005.
4. S. Chien, V. J. Tsotras, and C. Zaniolo. Efficient schemes for managing multiversionxml documents. *VLDB J.*, 11(4), 2002.
5. F. Currim, S. Currim, C. Dyreson, and R. T. Snodgrass. A Tale of Two Schemas: Creating a Temporal Schema from a Snapshot Schema with $\tau$XSchema. In *Proc. of EDBT*, pages 348–365, Heraklion, Greece, 2004.
6. C. De Castro, F. Grandi, and M. R. Scalas. Semantic interoperability of multitemporal relational databases. In *Proc. of ER*, 1993.
7. J. Van den Bercken, B. Blohsfeld, J. P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. Xxl - a library approach to supporting efficient implementations of advanced database queries. In *Proc. of VLDB*, pages 39–48, 2001.
8. R. T. Snodgrass et al. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishing, New York, 1995.
9. D. Gao and R. T. Snodgrass. Temporal slicing in the evaluation of xml queries. In *Proc. of VLDB*, pages 632–643, Berlin, Germany, 2003.
10. F. Grandi and F. Mandreoli. Temporal modelling and management of normative documents in xml format. *Data Knowl. Eng.*, 54(3), 2005.
11. H. Jiang, W. Wang, H. Lu, and J. Xu Yu. Holistic twig joins on indexed xml documents. In *Proc. of VLDB*, pages 273–284, 2003.
12. A. O. Mendelzon, F. Rizzolo, and A. A. Vaisman. Indexing temporal xml documents. In *Proc. of VLDB*, pages 216–227, 2004.
13. T. Bach Pedersen, C. S. Jensen, and C. E. Dyreson. Extending practical preaggregation in on-line analytical processing. In *Proc. of VLDB*, pages 663–674, 1999.
14. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proc. of ACM SIGMOD*, 2001.
15. D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *Proc. of ICDE*, pages 103–114, 2002.