Native Temporal Slicing Support for XML Databases

F. Mandreoli*, R. Martoglia and E. Ronchetti

DII - Università di Modena e Reggio Emilia, I-41100 Modena, Italy

Abstract: XML databases, providing structural querying support, are becoming more and more popular. As we know, XML data may change over time and providing an efficient support to queries which also involve temporal aspects is still an open issue. In this paper we present our native Temporal XML Query Processor, which exploits an ad-hoc temporal indexing scheme relying on relational approaches and a technology supporting temporal slicing. As we show through an extensive experimental evaluation, our solution achieves good efficiency results, outperforming stratum-based solutions when dealing with time-related application requirements while continuing to guarantee good performance in traditional scenarios.

Key Words: XML, native temporal XML query processor, temporal index, holistic twig join.

1. INTRODUCTION

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 XML data not only structurally, but also temporally. Recent works on this topic include [1-4].

The key to supporting most temporal queries in any language is to time-slice 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. The paper [2] has the merit of having been the first to raise the temporal slicing issue in the XML context, where timestamps are distributed throughout XML documents. The proposed solution relies on a *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 present our *native* solution to temporal slicing [5] and we compare it in detail with the traditional stratum approaches. Our idea is to propose the changes that a “conventional” XML pattern matching engine would need to be able to slice time-varying XML documents. In this way, 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, at the same time, we can freely extend them to become temporally aware. In particular, we exploit a novel temporal indexing scheme, which extends the inverted list technology proposed in [6] in order to allow the storing of time-varying XML documents, and a flexible technology supporting temporal slicing, consisting in alternative solutions relying on the holistic twig join approach [7], 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 and include the introduction of novel algorithms and the exploitation of different access methods. The benefits of our native approach over the stratum one are manifold: the native approach is able to access and retrieve only the strictly necessary data and there is no need to retrieve whole XML documents and build space-consuming structures such as DOM trees. Thus, main memory space requirements, I/O and CPU costs can be significantly limited.

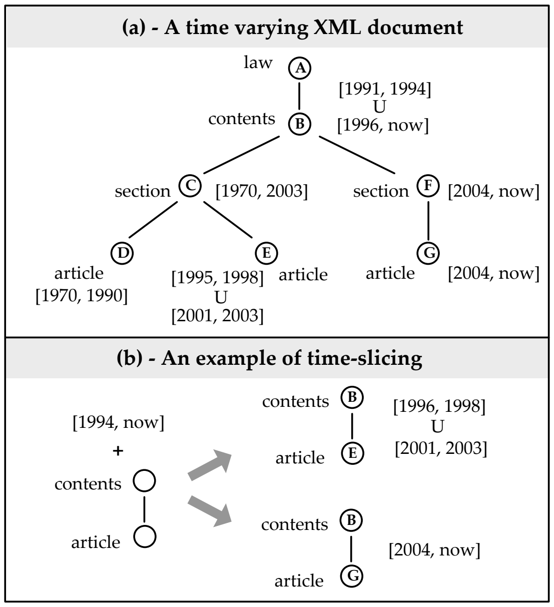


Fig. (1). Reference example.

This paper is organized as follows: we begin by analyzing the temporal slicing problem, comparing the stratum and native approaches in Section 2. In Section 3 we describe our native proposal’s indexing scheme and technology more in

*Address correspondence to this author at the DII - Università di Modena e Reggio Emilia, I-41100 Modena, Italy;
E-mail: federica.mandreoli@unimore.it

detail. Finally, Section 4 presents a range of experimental results and Section 6 concludes the paper.

2. TEMPORAL SLICING: STRATUM VS. NATIVE APPROACH

Let us start by clarifying the concepts of temporal querying on multiversion XML documents and, in particular, of temporal slicing. 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 [1]. Fig. (**1-a**) shows the tree representation of a time-varying XML document, which will serve as our reference example, taken from a legislative repository of norms. Data 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 temporal window and which matches with a given XML query twig pattern. The resulting slice consists of nodes that: (i) satisfy the query nodes predicates; (ii) are structurally consistent (i.e. parent-child and ancestor-descendant relationship are satisfied); (iii) are temporally consistent (i.e. the intersection of their lifetime is non-empty and contained in the temporal window). Fig. (**1-b**) shows the output of the temporal slicing example of our reference time-varying XML document for the period [1994,*now*].

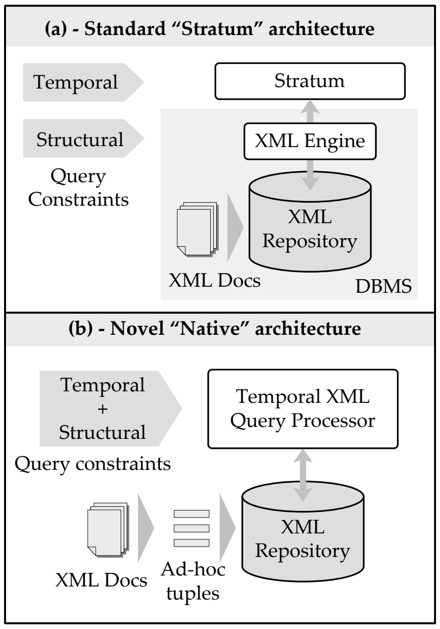


Fig. (2). "Stratum" versus "Native" architectures.

Supporting temporal slicing is the key to supporting most temporal querying applications. In the last years, different proposals have been made for querying the temporal aspect of XML data by means of temporal slicing. In [2], the authors suggest a stratum-based implementation to exploit the availability of XQuery engines. Indeed, stratum architectures are quite popular, however they not always deliver the desired level of performance. For instance, even if in [2] different optimizations of the initial time-slicing approach are proposed, the solution results in long XQuery programs also for simple temporal queries and postprocessing phases in order to coalesce the query results. Let us now analyze the features of a stratum architecture, comparing it with a native one.

As shown in Fig. (**2-a**), a traditional stratum architecture relies on two different components: A standard XML engine offering XML document management facilities and handling structural constraints, and a software stratum that is built on top of the former to handle the temporal aspects. The experimental results of an implementation of the stratum approach show a postprocessing behavior that is linear with the number of the documents involved in the results [3,8]. Moreover, a standard XML 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.

Instead, a native approach, such as the one we present in this paper, relies on a novel architecture (shown in Fig. 2-b). It is composed of a Temporal XML Query Processor designed on purpose, which is able to manage the XML data repository and to provide all the structural and temporal query facilities in a single component. Differently from the stratum approach, where temporal constraints are processed separately, all the structural and temporal constraints are simultaneously handled by the Temporal XML Query Processor. Such a component stores the XML norms not as entire documents but by converting them into a collection of ad-hoc temporal tuples, representing each of its multi-version parts. The benefits of the native approach over the stratum one are manifold: By querying ad-hoc and temporally-enhanced structures (which have a finer granularity than entire documents), the native approach is able to access and retrieve only the strictly necessary data. Then, only the parts which are required and which satisfy the temporal constraints are used for the reconstruction of the retrieved documents and there is no need to retrieve whole XML documents and build space-consuming structures such as DOM trees. Native solutions have been also proposed in [9], which introduces techniques for storing and querying multiversion XML documents, and in [4], where the authors propose an approach for evaluating TXPath queries integrating the temporal dimension into a path indexing scheme. However, these approaches show large overheads when "conventional" queries involving structural constraints and spanning over multiple versions are submitted to the system, since query processing requires the full navigation of the document collection structure and the execution of binary joins between them. Further, the main memory representation of the indices is very large (more than 10 times the size of the original documents in [4]. In the following section, we present in detail our native solution, which fully supports temporal slicing and tries to overcome these shortcomings, providing good querying performance both in temporal and traditional scenarios.

3. PROVIDING A NATIVE SUPPORT FOR TEMPORAL SLICING

Existing work on "conventional" XML query processing (see, for example, [6] shows that capturing the XML document structure using traditional indices is a good solution. 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 [6] such that time-varying XML databases can be implemented, and in alternative changes to the holistic twig join technology [7] in order to efficiently support the time-slice operator in different scenarios.

3.1. The Temporal Indexing Scheme

The indexing scheme described in [6] is an extension of the classic inverted index data structure in information retrieval which maps elements and strings to inverted lists. The position of an element occurrence is represented in each inverted list as a tuple (DocId, LeftPos:RightPos, LevelNum) where (a) DocId is the identifier of the document, (b) LeftPos and RightPos are the positions of 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 [6].

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. Our proposal adds time to the interval-based indexing scheme by substituting the inverted indices in [6] 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, 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, 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.

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 whole timeline of each temporal dimension on which it has not been defined. Fig. (**3**) 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. (**4**). 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 $q_1, \dots, q_n$, the algorithm presented in [7] works on the inverted indices $I_{q_1}, \dots, I_{q_n}$ (level L0 in Figure) and builds solutions from the stacks $S_{q_1}, \dots, S_{q_n}$ (level L2 in Figure).

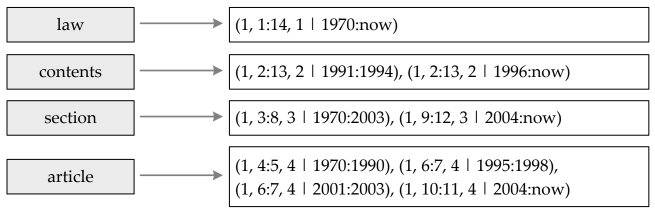

law
(1, 1:14, 1 | 1970:now)
contents
(1, 2:13, 2 | 1991:1994), (1, 2:13, 2 | 1996:now)
section
(1, 3:8, 3 | 1970:2003), (1, 9:12, 3 | 2004:now)
article
(1, 4:5, 4 | 1970:1990), (1, 6:7, 4 | 1995:1998),
(1, 6:7, 4 | 2001:2003), (1, 10:11, 4 | 2004:now)

Fig. (3). The temporal inverted indices for the reference example.

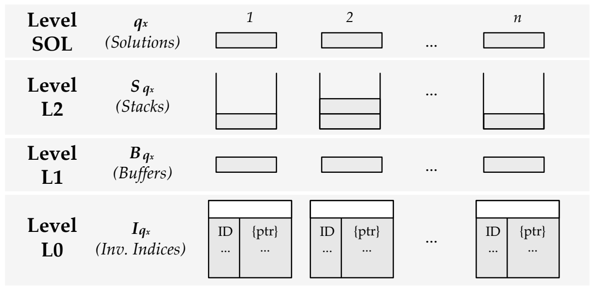


Fig. (4). The basic holistic twig join architecture.

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 space a compact encoding of partial and total answers to the query twig pattern. The skeleton of the holistic twig join (HTJ from now on) algorithm is the following:

```
While there are nodes to be processed
(1) Choose the next node $n_{\bar{q}}$
(2) Apply the deletion policy
(3) Push the node $n_{\bar{q}}$ into the pertinence stack
$S_{\bar{q}}$
(4) Output solutions
```

At each iteration the algorithm identifies the next node to be processed. To this end, for each query node q, at level L1 there is the node in the inverted index I_q with the smaller LeftPos value and not yet processed. Among those, the algorithm chooses the node with the smaller value, let it be $n_{\bar{q}}$. Then, it removes 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 algorithm outputs these answers.

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. Thus the holistic twig join algorithms continue to work, as they are responsible for the structural consistency of the slices and provide 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. The performances of this first solution are less than optimal, since 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. To the light of these facts, a smart management of the temporal consistency aspects is needed.

3.3. Managing Temporal Consistency

Temporal consistency considers two aspects: The intersection of the involved tuples' lifetimes must be non-empty (*non-empty intersection constraint*) and it must be contained in the temporal window (*containment constraint*). We devised alternative solutions which rely on these 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.

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},... ,I_{q_k} the inverted indices of the ancestors of q and $TP_{q_i}=\sigma_{\texttt{LeftPos<L}}(I_{q_i})$|TempPer, for i∈[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},... ,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 indices. However, looking for such tuples in the three levels of the architecture would be quite computationally expensive. Thus, we proceed in two ways: We exploit an enhanced buffer loading algorithm (`Load` algorithm in the following) which allows us to look only at the stack level and we associate a temporal pertinence to each stack (*temporal stack*), thus avoiding to access the temporal pertinence of the tuples contained in the stacks. 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 multidimensional period P_q containing the union of the temporal pertinence of the tuples in the stack S_q.

The intuition behind the Load algorithm [5] is to avoid loading the temporal tuples encoding a node 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. In this way, a tuple (*D*,*L*:*R*,*N*|*T*) in

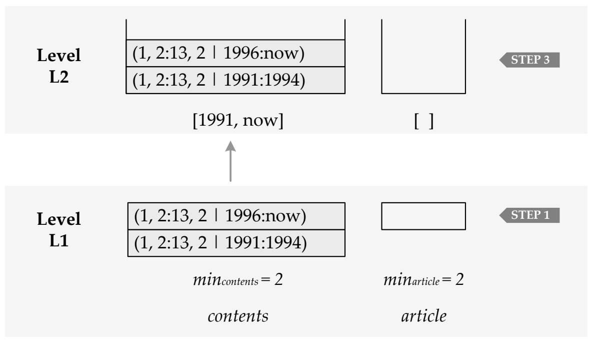


Fig. (5). Levels L1 and L2 during the first iteration.

B_q will belong 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,... ,q_k of *q* is empty. For instance, at the first iteration of the HTJ algorithm applied to the reference example, step 1 and step 3 produce the situation depicted in Fig. (**5**). 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, thanks to the above consideration, we can state that it will belong to no slice.

We can exploit the non-empty intersection constraint to prevent the insertion of useless nodes into the stacks by acting at level L1 or L2 of the architecture. At level L2 we act at step 3 of the HTJ algorithm by simply avoiding pushing into the stack S_q each temporal tuple (*D*,*L*:*R*,*N*|*T*) encoding the next node to be processed. At level L1, instead, we avoid loading in any buffer B_q each temporal tuple encoding n_q which will belong to no slice. In this case, the only loaded tuples will be those having the minimum LeftPos greater than the one of the last processed node and whose TempPer intersects the period of the ancestor stacks. To this purpose, our solution uses time-key indices combining the LeftPos attribute with the attributes $From_j$:To_j of TempPer representing one temporal dimension in order to improve the performances of range-interval selection queries on the inverted indices. In order to be able to use simple B+-trees, which cluster data primarily on a single attribute, we performed an attribute concatenation and we built B+-trees that cluster first on the LeftPos attribute, then on the end time To_j and finally on the start time $From_j$ of the interval. Further, in the experimental evaluation section we also compare this kind of access method with one based on the Multiversion B-tree (MVBT) [10]. As to the containment constraint, the following proposition holds:

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, where 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, for selecting the relevant tuples.

3.4. 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). 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 native XML query processor supporting temporal slicing, comparing it with the stratum implementation presented in [3] and showing its behavior in different execution scenarios.

The document collection follows the structure of the documents used in [3], 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, each one in 2-3 versions composed by the union of 1-2 distinct periods.

Experiments were conducted on a reference collection, 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 paper 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.

4.1. 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.

4.1.1. 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 returned solutions are 5584.

Table 1. Evaluation of Computation Scenarios with TS1

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 %

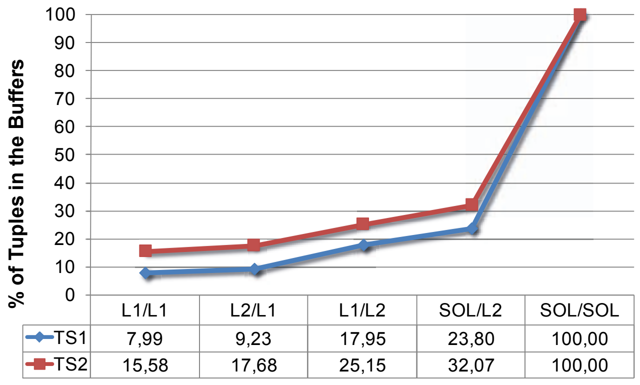
Table **1** shows the performance of each scenario when executing TS1. In particular, from the left: The execution time, the percentage of potential solutions at level SOL that are not temporally consistent and, in the last two columns, the percentage of tuples that are put in the buffers and in the stacks w.r.t. the total number of tuples involved in the evaluation.

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. 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 [5] for an in-depth analysis). 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.

4.1.2. Native vs Stratum Comparison

After having measured the behavior of the native implementation in the default setting, we wanted to compare its performance to the one obtainable through a standard stratum approach. In order to do that, we performed additional tests using an available implementation of a temporal-aware stratum-based engine.

In general, as we saw in the past sections, stratum approaches require two distinct phases in order to provide the



	L1/L1	L2/L1	L1/L2	SOL/L2	SOL/SOL
TS1	7,99	9,23	17,95	23,80	100,00
TS2	15,58	17,68	25,15	32,07	100,00

Fig. (6). Perc. of Non-Consistent Solutions.

final results since they handle structural and temporal constraints in separate components. In the first phase, all the whole documents satisfying the structural constraints are retrieved, then from the DOM representation the portions of each of these documents that do not verify the temporal constraints are pruned out in a post-processing phase.

From the tests we performed, we saw that our stratum processor was able to perform the first phase of the default setting in nearly 20 seconds, which is more than 7 times the time required by our novel XML query processor. Further, and this is typical of most stratum implementations, the postprocessing phase was linear with the number of the documents retrieved; in our case, it processed nearly 10 documents per second.

Summing up, stratum approaches have many shortcomings w.r.t. native ones. In particular:

- stratum-level constraints have to be evaluated in a second moment, so that, even if a granularity finer than the whole document is adopted, a large quantity of additional information has to be retrieved anyway for their evaluation;
- for the same reason, the Stratum is not able to optimize query execution with an access plan filtering the data always on the most selective constraints first;
- the structure of a document is analyzed twice: a first time in the XML engine when the structural constraint is resolved and a second time in the Stratum to correctly evaluate the other constraints.

On the other hand, the Native system is able to deliver a fast and reliable performance in all cases, since it practically avoids the retrieval of useless document parts and is not as demanding as the Stratum in terms of main memory space. Notice that this property is also very promising towards future extensions to cope with concurrent multi-user query processing, since memory requirements are not crucial for performance.

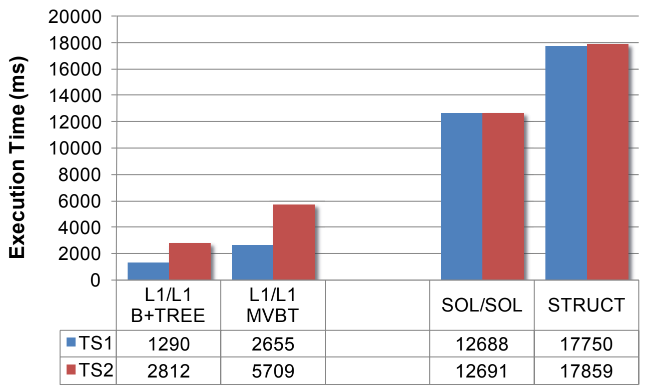
From these and further tests we performed, we can state that a native implementation such as ours generally outperforms stratum performance in most temporal settings. Further, the native implementation required less than 5% of main memory of the DOM-based approach, typically used in stratum implementations.

4.1.3. 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. Fig. (**6**) shows the percentage of read tuples of TS2 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 (the trends in execution time show similar behaviours). Further, in the SOL/SOL scenario both queries have the same number of tuples and execution time in the buffers because no selectivity is applied at the lower levels.

4.1.4. Comparison with MVBT and purely structural techniques.

In Fig. (**7**) 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 [11]. The last comparison involves the holistic twig join algorithms applied on the original indexing scheme proposed in [6] 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 Fig. (**7**) 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.

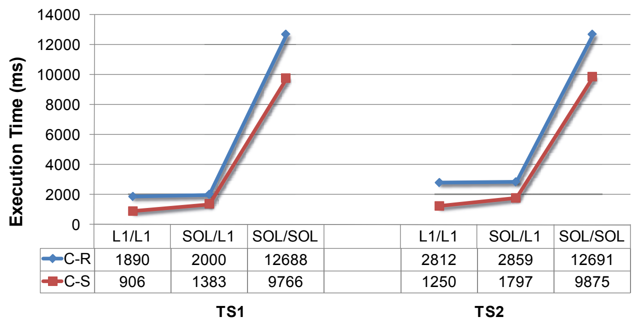


	L1/L1 B+TREE	L1/L1 MVBT		SOL/SOL	STRUCT
TS1	1290	2655		12688	17750
TS2	2812	5709		12691	17859

Fig. (7). MVBT and structural approach performances.

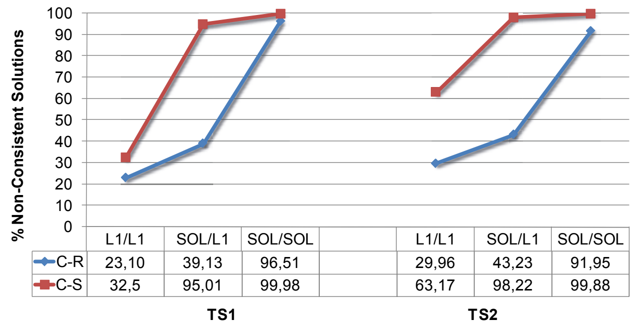
4.1.5. Evaluation on Differently Distributed Collections

We also considered the performance of our XML query processor on another collection (C-S) of the same size of the reference one (C-R), but that is characterized by temporally scattered nodes. Figs. (**8** and **9**) show the execution time and the number of temporally inconsistent potential solutions of TS1 and TS2 on both collections. The execution time of scenarios L1/L1 and SOL/L1, depicted in Fig. (**8**), 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**). 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.



	L1/L1	SOL/L1	SOL/SOL		L1/L1	SOL/L1	SOL/SOL
C-R	1890	2000	12688		2812	2859	12691
C-S	906	1383	9766		1250	1797	9875

Fig. (8). Comparison between the two collections C-R and C-S: Execution time.

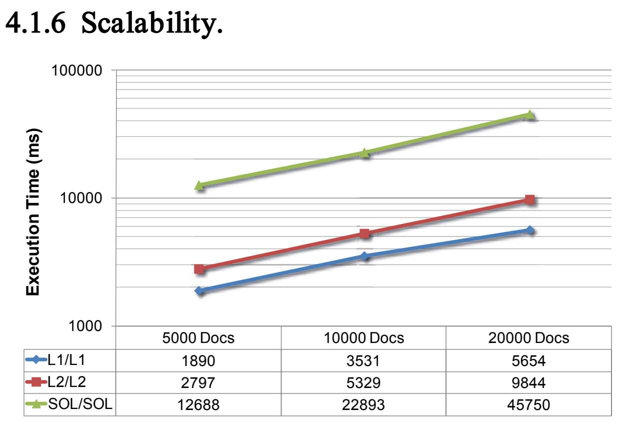


	L1/L1	SOL/L1	SOL/SOL		L1/L1	SOL/L1	SOL/SOL
C-R	23,10	39,13	96,51		29,96	43,23	91,95
C-S	32,5	95,01	99,98		63,17	98,22	99,88

Fig. (9). Comparison between the two collections C-R and C-S:Percentage of Non-Consistent Solutions

4.1.6. Scalability

Fig. (**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.



	5000 Docs	10000 Docs	20000 Docs
L1/L1	1890	3531	5654
L2/L2	2797	5329	9844
SOL/SOL	12688	22893	45750

Fig. (10). Scalability results for TS1.

5. DISCUSSION

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 [1-4], where the history of changes XML data undergo is represented into a single document from which versions can be extracted when needed. In [1], 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 [2] presents a temporal XML query language, τXQuery, 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 [4] extends XPath for supporting temporal queries. Finally, the main objective of the work presented in [3] 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, [2] 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 [4,9]. The paper [9] 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 [4] 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 [6] 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. [12]) also using indices [13]. 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.

6. CONCLUSION

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 [7], we proposed new flexible technologies consisting in alternative solutions and extensively experimented them in different settings. The resulting Temporal XML Query Processor overcomes many of the shortcomings of stratum implementations and its efficiency is quite encouraging and induces us to continue in this direction.

REFERENCES

[1] F. Currim, S. Currim, C. Dyreson, and R. T. Snodgrass, "A tale of two schemas: Creating a temporal schema from a snapshot schema with τXSchema", in Proceeding of Extending Database Technology, Heraklion, Greece, 2004, pp. 348-365.

[2] D. Gao, and R. T. Snodgrass, "Temporal slicing in the evaluation of xml queries", in Proceeding of Very Large Data Bases, Berlin, Germany, 2003, pp. 632-643.

[3] F. Grandi, F. Mandreoli, and P. Tiberio, "Temporal modelling and management of normative documents in XML format", *Data Knowledge Engineering*, vol. 54(3), pp. 327-354, September 2005.

[4] A. O. Mendelzon, F. Rizzolo, and A. A. Vaisman, "Indexing temporal xml documents", in Proceedings of Very Large Data Bases, 2004, pp. 216-227.

[5] F. Mandreoli, R. Martoglia, and E. Ronchetti, "Supporting Temporal Slicing in XML Databases", in Proceedings of Extending Database Technology, 2006, pp. 295-312.

[6] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman, "On supporting containment queries in relational database management systems", in Proceeding of ACM SIGMOD, 2001, pp. 902-919.

[7] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal XML pattern matching", in Proceedings of the ACM SIGMOD, 2002, pp. 310-321.

[8] F. Grandi, F. Mandreoli, R. Martoglia, and M. R. Scalas, "Efficient management of multi-version xml documents for e-government applications", in Proceedings of Web Information Systems and Technology, Miami, FL, 2005.

[9] S. Chien, V. J. Tsotras, and C. Zaniolo, "Efficient schemes for managing multiversion XML documents", *Very Large Data Bases Journal*, vol. 11(4), pp. 902-919, 2002.

[10] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion b-tree", *Very Large Data Bases J.*, vol. 5(4), pp. 264-275, December 1996.

[11] 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 Proceedings of Very Large Data Bases, 2001, pp. 39-48.

[12] T. Bach Pedersen, C. S. Jensen, and C. E. Dyreson. "Extending practical pre-aggregation in on-line analytical processing", in Proceedings of Very Large Data Bases, 1999. pp. 663-674.

[13] D. Zhang, V. J. Tsotras, and B. Seeger, "Efficient temporal join processing using indices", in Proceedings of International Conference of Data Engineering, 2002, pp. 103-114.

Received: March 31, 2008 Revised: April 04, 2008 Accepted: April 28, 2008