Indexing Temporal XML Documents

Alberto O. Mendelzon
University of Toronto
Department of Computer Science
mendel@cs.toronto.edu

Flavio Rizzolo
University of Toronto
Department of Computer Science
flavio@cs.toronto.edu

Alejandro Vaisman
Universidad de Buenos Aires
Departamento de Computación
avaisman@dc.uba.ar

Abstract

Different models have been proposed recently for representing temporal data, tracking historical information, and recovering the state of the document as of any given time, in XML documents. We address the problem of indexing temporal XML documents. In particular we show that by indexing *continuous paths, i.e.* paths that are valid continuously during a certain interval in a temporal XML graph, we can dramatically increase query performance. We describe in detail the indexing scheme, denoted TempIndex, and compare its performance against both a system based on a non-temporal path index, and one based on DOM.

1 Introduction

The topic of representing, querying and updating temporal information in XML documents has been receiving increasing attention from the database community, leading to proposals aimed at defining, querying and managing temporal XML documents, *i.e.* XML documents that can be navigated across time.

In a separate paper [23], we propose a model for temporal documents and a query language called TXPath. TXPath extends XPath [25] for supporting temporal queries (*i.e.* queries over temporal XML documents.) This abstract data model represents the temporal document as a data graph which has time interval information on its paths. Navigating this temporal data graph is a key part of the TXPath query evaluation. However, simply scanning the whole document in search of those paths that satisfy a given temporal query is highly expensive.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

Several index structures have been proposed in order to optimize path query evaluation over non-temporal data graphs. Some of the more recent works on path indexing in the XML context include [12, 18, 7, 17, 14]. Most of these indexing schemes keep record of the paths in the XML data by summarizing label path information. Although indexing label paths on temporal documents helps reduce the search space, our experiments show that computing paths within a given time interval is quite expensive even in the presence of traditional path indexes. One possible solution is to integrate the temporal dimension into the indexing scheme in order to obtain better performance. Our proposal, denoted TempIndex, accomplishes this integration by summarizing label paths together with temporal intervals and *continuous paths* (*i.e.* paths that are valid continuously during a certain interval.)

1.1 Temporal XML documents

In Section 3 we review the temporal XML model that we use. We give an example here. The graph depicted in Figure 1 is an abstract representation of a temporal XML document for a portion of the NBA[1] database. The league is composed of franchises, which maintain teams, such that each team has a set of players that may change over time. Some franchises may have players directly associated to them, not included in teams. The database also records some statistics for each player. For instance, in this database, node 16 represents a player (McGrady), playing for the Toronto Raptors between instants '0' and '20'. After that, he played for the Orlando Magic (represented by node 2), from instant '21' to the present time (note the edge between nodes 2 and 16.) Notice that in spite of the change of franchise, there is only one node for McGrady. Thus, regardless of the franchise he played for, the graph shows that he scored eleven goals between instants '0' and '30', and twelve goals from instant '31' to the present time. This information is encapsulated

[1]National Basketball Association, a professional basketball league

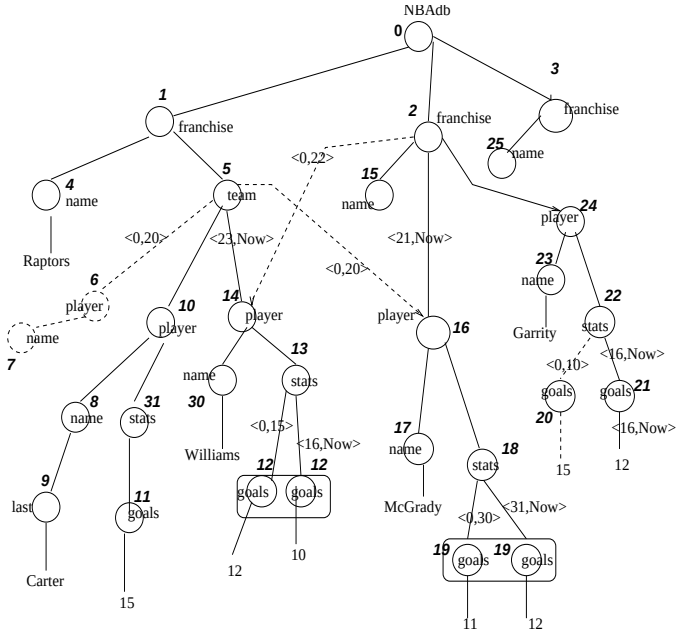

Figure 1: Example database

in a sequence that we denote *versioned* node (see Section 3), represented by the box enclosing the two nodes labeled '19'.

1.2 Contributions

In this paper we describe in detail the TempIndex indexing scheme. In addition, we present the results of our experiments, which show that an index summarizing temporal intervals and continuous paths clearly outperforms traditional path indexes on temporal query evaluation.

The remainder of the paper is organized as follows: in Section 2 we comment on previous efforts in XML path indexing and temporal XML. In Section 3 we review the main features of the data model and the TXPath query language. In Section 4 we present the details of the proposed indexing scheme while in Section 5 we show how a TXPath query is processed using TempIndex. Finally, Section 6 discusses the results of our experiments. We conclude in Section 7.

2 Related Work

Index structures for XML data have been proposed in recent years in order to optimize path query evaluation. Most of these indexing schemes keep record of the paths in the XML data by summarizing path information in different ways. Examples of such index structures are dataguides [12], 1-indexes and T-indexes [18], and more recently, Index Fabric [7], XISS [17], ToXin [20], F&B-Index and F+B-Index [14], and A(k)-index [16]. On a different vein, *adaptive indexes* are based on query workloads that may change over time, such as APEX [6] and D(k)-index [19]. Finally, Kaushik *et al* [15] discuss fast updating of structure indexes.

Dataguides are a summary of the path structure of the database in which every label path starting at the root appears exactly once. The nodes in the data graph are grouped into sets according to the label paths they belong to (each node may appear more than once in the index). The 1-index, T-index, F&B-Index and F+B-Index, on the other hand, partition the data graph nodes into equivalence classes so that each node appears only once. The partition is computed in different ways: based on the label paths that reach the nodes (1-index) or further refined into smaller classes according to the label paths of any length (F&B-Index) or of a fixed length (F+B-Index) that leave from the nodes. The size of the index can also be reduced by indexing only the class of paths specified by a given path template (T-index), or making the index approximate for paths longer than a given k (A(k)-index). All these indexes either store a limited class of paths or the number of equivalence classes grows to a point in which evaluating generic XPath queries is no longer efficient. Other two approaches (XISS and ToXin) use separate structures for storing nodes. Both implement different join algorithms to efficiently reconstruct paths of any length. In addition to that, ToXin keeps a dynamic schema of the document for query optimization. Index Fabric, in a completely different approach, summarizes paths and data values together, and encodes them as strings.

In the temporal XML field, many efforts [1, 8, 3, 4] have proposed data models and query languages for representing the histories of XML documents. Most of them create a new physical version each time an update occurs, leading to large overheads when processing temporal queries that span multiple versions. A *version index* for managing multiple versions of XML documents was proposed by Chien *et al* [5]. The TXPath temporal data model, on the other hand, maintains a single temporal document from which versions can be extracted when needed. Gergatsoulis and Stavrakas [11] introduced a model for representing changes using an extension to XML denoted MXML (Multidimensional XML), where dimensions are applied to elements and attributes.

Closer to TXPath ideas, Gao *et al* [9, 10] introduced an extension to XQuery, called τXQuery, that supports valid time while maintaining the data model unchanged. Queries are translated into XQuery, and evaluated by an XQuery engine. Even for simple temporal queries, this approach results in long XQuery programs. Moreover, translating a temporal query into a non-temporal one makes it more difficult to apply query optimization and indexing techniques particularly suited for temporal XML documents.

In this work we will take advantage of the structure of the temporal XML document, showing that it is pos-

sible to index temporal continuous paths rather than nodes, enhancing query performance dramatically.

3 Temporal XML

A *temporal XML document* is a directed labeled graph with different kinds of nodes: the *root* of the document, denoted r, such that r has no incoming edges; *Value nodes*, representing text or numeric values; *Attribute nodes*, labeled with the name of an attribute, plus possibly one of the 'ID' or 'REF' annotations; and *Element nodes,* labeled with an element tag, and containing outgoing links to attribute nodes, value nodes, and other element nodes. Each node is uniquely identified by an integer, the *node number,* and is described by a string, the *node label.* Edges in the document graph can be either *containment edges* or *reference edges*. Containment edges connect element, attribute or value nodes, while reference edges represent IDREF to ID references. Each edge e is labeled by a time interval T_e that represents the *valid* time of the edge, *i.e.* the interval during which the edge was valid. Time is discrete, with instants represented by positive integers. The *lifespan* of a node is the union of all the containment edges incoming to the node. If an edge e is labeled with a temporal label T_e, we will use $T_e.TO$ and $T_e.FROM$ to refer to the endpoints of the interval T_e. Temporal XML documents also support the concept of *versioned nodes,* which encapsulate a sequence of *consecutive* element or attribute nodes (of type other than ID or REF).

A temporal XML document must verify some consistency conditions. The key ones are: the union of the temporal labels of the containment edges outgoing from a node is contained in the lifespan of the node; the temporal labels of the containment edges incoming to a node must be consecutive; and, for any time instant t, the sub-graph composed of all the containment edges e such that $t \in \mathrm{T}_e$ is either empty or a tree with root r (we call such a subgraph a *snapshot* of the document at time t.)

Example 1 *In Figure 1 the fact that McGrady played for the Orlando Magic from instant '21' to the current time, is represented by the containment edge* $e(2, 16)$. *The lifespan of node '16' is the union of the elements [0,20] (the temporal label of the incoming node edge from node 5) and [21,Now] (the label of the containment edge incoming from node 2). The boxes in bold line represent versioned nodes, composed in this case by two nodes of type* goals*, associated with the element nodes* Stats*. To simplify the figures, we omit all temporal labels of the form* $[t_0, Now]$*, and represent containment edges currently valid by solid lines; other containment edges are represented by dashed lines.*

There are different ways of mapping the abstract graph to an XML document [23]. For the rest of this paper, except for the experimental results, it does not matter which representation is used. We sketch below the representation we used for the experiments. A node n is physically nested within its "oldest" parent. If n was contained in some other parent p during interval T, an element with the same tag as n, annotated with T, and pointing to the ID for node n, is nested within node p. Below, we show a portion of the XML document resulting from mapping the graph in Figure 1:

```
<NBAdb>
  <franchise ID='1'[0,Now]>
    <name[0,Now]>Raptors</name>
    <team[0,Now] ID = '5' >
      <player[23,Now] IN = '14'/>
      ...
      <player[0,20] ID='16'>
        <name[0,Now]>Tracy McGrady</name>
        <stats[0,Now]>
          <SEQUENCE>
            <goals[0,30]>11</goals>
            <goals[31,Now]>12</goals>
          </SEQUENCE>
        </stats>
      </player>
```

The inclusion of the element `<player [23,Now] IN = '14'/>` within node 5 means that the player represented by node 14, whose oldest parent is node 2, was contained in node 5 between 23 and Now. For the sake of clarity we use a simplified syntax for the XML documents. For example, `<franchise ID = '1' [0,Now]>` would actually read `<franchise ID='1' Time:FROM = '1999-01-01' Time:TO = 'Now'>`.

TXPath Overview

The TXPath query language extends XPath 2.0 [25] with temporal features. In non-temporal XPath 2.0, the meaning of a path expression is the sequence of nodes at the end of each path that matches the expression. In TXPath, the meaning is a sequence of (node,interval) pairs such that the node has been continuously at the end of a matching path during that interval. To make this precise, we define the notion of *continuous path*, which we will be using throughout the paper, and *maximal continuous path.*

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

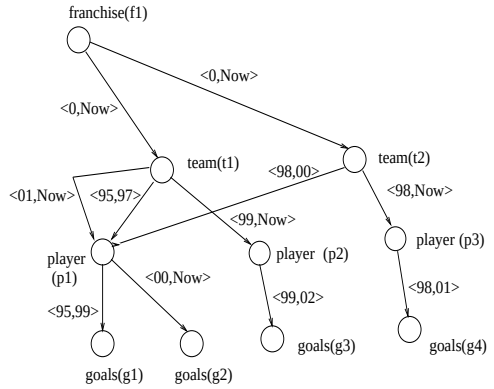

Figure 2: Maximal Continuous Path

consecutive intervals T_i such that there is a continuous path from n_1 to n_k with interval T_i.

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

Figure 3 shows the semantics of the most common TXPath constructs, adapting the formal XPath semantics introduced by Wadler [24]. The meaning of a TXPath expression is specified with respect to a *context pair* (node,interval). We define three semantic functions: $\mathcal{S}, \mathcal{Q}$ and $\mathcal{Q}_T$ such that $\mathcal{S}[\![p]\!]x$ denotes the sequence of pairs (node,interval) (or values, as we will see below) selected by pattern p when x is the context pair. The boolean expression $\mathcal{Q}[\![q]\!]x$ denotes whether the qualifier q is satisfied when the context pair is x. Finally, another boolean expression $\mathcal{Q}_T[\![q_T]\!]x$ denotes whether a temporal condition q_T is satisfied.

In order to give the flavor of the language, let us show the query *"Player nodes for players with the Toronto Raptors on October 10th, 2001"* in TXPath:

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

Assuming that the date October 10th, 2001 is represented by instant 15, the result is the sequence $\{6, [0, 20]; 10[0, Now]; 16, [0, 20]\}$. Note that the order of the answer corresponds to the *document* order at the time asked for in the query. If the query had not asked for a particular instant, the result would have been listed in arbitrary order.

Document order

In a non-temporal XML document, there is a total order between the nodes. A temporal document does not necessarily impose a total order among its nodes, but for any instant t there must be a total order, denoted $<_t$, among the nodes of each snapshot $D(t)$ of document D at time t. In general, for any pair of nodes n_1 and n_2, we may have $n_1 <_{t_1} n_2$, and $n_2 <_{t_2} n_1$, in two different instants t_1 and t_2. However, we can show that there is an interval during which the relative order between n_1 and n_2 does not change. If I_1 is the interval on a continuous path from the root to n_1, and similarly I_2 for n_2, then the ordering between n_1 and n_2 is the same for any instant t in the interval $I_1 \cap I_2$. This is formalized in the following proposition.

$\mathcal{S}$

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

$\mathcal{Q}$

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

$\mathcal{Q}_T$

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

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

Figure 3: Formal semantics of TXPath

Proposition 1 *Let D be a temporal XML document; n_1 and n_2 two nodes in D; $p_1 = (r, \ldots, n_1, I_1)$ and $p_2 = (r, \ldots, n_2, I_2)$ two continuous paths to n_1 and n_2 with intervals I_1 and I_2, respectively; then, either $n_1 <_t n_2$ for every $t \in I_1 \cap I_2$, or $n_2 <_t n_1$ in every such t.*

4 Temporal Indexing Scheme

As we mentioned in Section 1, efficiently querying temporal XML documents requires the ability to find the paths in the graph that were valid at a given time (*i.e.* the *continuous paths* in the document.) This ability is not provided by traditional path indexes. Our pro-

posal adds the time dimension to path indexing by indexing *continuous paths* to element or value nodes.

The standard notion of label paths can be easily extended to continuous paths. Let $p = (n_1, \ldots, n_k, T)$ be a continuous path with interval T. The *label path of p*, denoted $\lambda(p)$ is the concatenation of the labels of the n_i in p.

Traditional path indices [12, 18] often define equivalence classes of nodes that are reachable from the root by a path with the same label. In TempIndex, we define equivalence classes of pairs $\langle node, interval \rangle$ such that for all the pairs $\langle n, I \rangle$ in a class, there is a continuous path from the root to n, with interval I and the same label. This classes are stored in tables called *cp* and *cp+value*, which we will define next.

Definition 2 (CP and CP+Value Tables) *Consider the set of all labels $\lambda(p)$ such that p is a continuous path from the root of document d to some node in d. For each string l in this set, let $[l]$ be the equivalence class of all pairs $\langle n, I \rangle$ such that n is a node in d, I is an interval, and there is a continuous path from the root of d to n with interval I and label l. For each class $[l]$ in d there is a* cp table *in which each tuple t has attributes* `parent`, `child`, `from` *and* `to` *such that there is a continuous path from the root of d to* `t.child` *with interval* `[t.from, t.to]` *via* `t.parent`. *When* `t.child` *has a value* `v` *associated to it, the cp table is called* cp+value table *and has an extra attribute named* `value`, *where* `t.value=v`. *Tuples in the cp and cp + value tables are sorted by* `child`.

In other words, each *cp* and *cp+value* table corresponds to a path label l and encodes the last edge of all the continuous paths in d labeled by l. The parent-child relationship contained in each tuple is used during query evaluation to traverse continuous paths with a given label and interval (see Subseccion 5 for more details). Figure 4 shows the *cp* table for the path */NBA/franchise/player/stats* and 5 shows the *cp+value table* for the path *NBA/franchise/player/name*.

Indexing intervals

Since cp's in the *cp* and *cp+value* tables are clustered by label and sorted by the `child` attribute, we need additional structures to index the intervals and to capture the node ordering at any given instant (as defined in Proposition 1). There are many proposals in the literature for indexing temporal intervals. Some of them are based on the methods proposed by Bozkaya *et al* [2] and Salzberg *et al* [21], where a B+ tree indexes the `FROM` value in the intervals being indexed, and each internal node is augmented with the information of the maximum `TO` value in an interval of the corresponding subtree. We propose a different scheme, embodied in

Parent	Child	From	To
14	13	0	22
16	18	21	Now
24	22	0	Now

Figure 4: *cp* table NBA/franchise/player/stats

Parent	Child	From	To	Value
16	17	21	Now	McGrady
24	23	0	Now	Garrity
14	30	0	22	Williams

Figure 5: *cp+value* table NBA/franchise/player/name

a set of tables called δ_k tables, that are based on the notion of *temporal depth*.

Definition 3 (Temporal Depth) *For each node n in d such that there exists a continuous path $cp = (r, \ldots, n, I)$ in d, $\delta(n, I) = length(cp)$ is a function called the* temporal depth *of n during the interval I. (Note that, due to the consistency conditions of Section 3, there is at most one continuous path with interval I from the root to each node n).*

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

Definition 4 (Node Validity) *A node n is* valid *at temporal depth k in an interval I iff there exists an interval I' such that $\delta(n, I') = k$ and $I \subseteq I'$.*

Thus, $\delta(n, I)$ defines an equivalence relation between the nodes in the temporal XML graph where for each pair $\langle n, I \rangle$ in a class the length of the continuous path from the root to n is the same. For each temporal depth k, we will define a table called δ_k table, listing the nodes that are valid at certain intervals and their relative order. These intervals are obtained by taking all the intervals that label some continuous path of length k and partitioning them as needed to obtain a set of pairwise-disjoint intervals. This is formalized with the notion of *interval partition*.

Definition 5 (Interval Partition) *The interval partition P of a set of intervals $I_1 \ldots I_n$ is the smallest set of intervals $\mathcal{P} = P_1 \ldots P_m$ such that all the P_i's in $\mathcal{P}$ are pairwise disjoint and $\mathcal{P}$ contains a partition of every interval I_j.*

Definition 6 (δ_k Tables) *For each temporal depth k in a document d there is a table called δ_k* table. *Each tuple* `t` *in a δ_k table has two temporal attributes,* `from`, `to`, *and a list-valued attribute* `valid`. *Let $I_1 \ldots I_n$ be all the intervals such that there is a cp of length k labeled by one of the I_j's, and $P_1 \ldots P_m$ be the interval partition of $I_1 \ldots I_n$. Each P_k is represented by a tuple* `t` *in δ_k. The* `t.valid` *attribute contains the*

From	To	Valid
0	19	$\{9, 11, 19\}$
20	22	$\{9, 11\}$
23	Now	$\{9, 11, 12\}$

Figure 6: δ_5 table

list of all nodes at temporal depth k that are valid in the interval `[t.from, t.to)`*. The nodes in* `t.valid` *are ordered by the order relation defined in the interval* `[t.from, t.to]`*. (Note that, according to Proposition 1, this order relation is always defined for all nodes in* `[t.from, t.to]`*). Tuples in the δ_k tables are indexed by* `from` *and* `to`*.*

For example, Figure 6 shows the δ_5 table for the running example.

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

Space Requirements

The size of the index is proportional to the number of cp's. Our experiments in Section 6 show that, for the NBA database, the number of cp's is about three times the number of nodes in the temporal graph. We support three types of updates, insertion, deletion and modification. When the XML graph is a tree, *i.e.* before any update is performed, for each edge in the temporal graph there is one tuple in the *cp/cp+value* tables. Furthermore, since there is only one interval of relevance, [0, Now], there is only one tuple t in each δ_k and the list of its valid nodes contains all nodes at temporal depth k. As updates are performed, the number of cp's in the document – and consequently the number of tuples in the tables – increases. The tables affected by an update are those that index descendants of a node at the update point, so the closer the update is to the root, the larger the increase in the index size. Occasionally, an update may also create a new partition in a δ_k table, in which case the nodes from the last partition that are still valid in the new partition have to be replicated.

There are several ways to reduce the space requirements for the index. In many applications, we expect most updates to occur close to the leaves, so that the size of the index will grow linearly in the size of the document. Our experiments so far confirm that expectation: the main-memory representation of TempIndex has a size comparable to that of the DOM representation (see Section 6) for both document sizes tested.

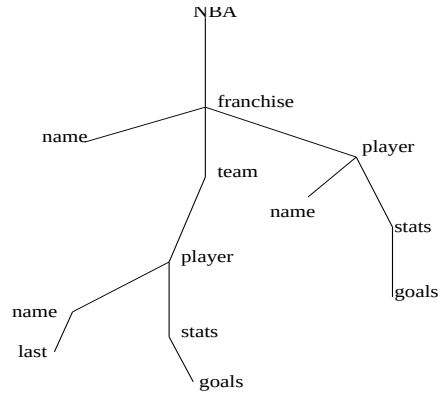

Figure 7: Temporal Schema

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

Finally, there is a lot of room for compression in the main memory TempIndex structures. For example, the Java `date` datatype that we are currently using consumes a lot more space than a typical application would need. We have made no attempt yet to optimize space usage by the index.

5 Query processing in TXPath

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

Before giving the query evaluation algorithms we will need the notion of *temporal schema*.

Definition 7 (Temporal Schema) *Consider the $[l]$ classes from Definition 2. The* temporal schema $S(d)$ *is a tree with a node for each class $[l]$. The root of the tree corresponds to the class for the root label. There is an edge from $[l_1]$ to $[l_2]$ in $S(d)$ iff there is an element name e in the document such that $l_2 = l_1.e$ (where "." is the concatenation symbol). Each node n in $S(d)$ has either a* cp *or a* cp+value *tables associated to it.*

We decompose each TXPath query into a sequence of calls to evaluation procedures called `navigate()`, `pathFilter()` and `tempFilter()`. Each procedure

receives as a parameter a set of temporal schema nodes N and a list of pairs (child, interval) from the cp/cp+value tables associated to the nodes in N. In addition, procedure `pathFilter()` receives a path expression and a value selection, `tempFilter()` a temporal predicate and `navigate()` a path expression between filters. All procedures return a set M of temporal schema nodes and a list of pairs (child, interval) from the cp/cp+value tables associated to the nodes in M. For simplicity, we consider in this discussion only filters that contain either path/value selections or temporal predicates, but not both. Based on these procedures, a query of the form:

```
PathExp1[PathExp2=Val][TempPred]...PathExpn
```

Will be decomposed as follows:

```
list.add(graph root, interval);
set.add(schema root);
navigate(set, list, PathExp1);
pathFilter(set, list, PathExp2, Val);
tempFilter(set, list, TempPred);
...
navigate(set, list, PathExpn);
```

Next, we will give the algorithms for `navigate()` and `pathFilter()`; the one for `tempFilter()` is similar.

5.1 Query Evaluation

Algorithm 1 (navigate) *Input/Output: nodeSet, pairList. Input: PathExpr, Value.*

1. *Compute the NFA P corresponding to PathExpr.*
2. *Determine the cp and cp+value tables that participate in the evaluation and navigate among them as follows:*
   - 2.1. *We view the temporal schema as an automaton T whose states correspond to the schema nodes. Each of T's transitions corresponds to an edge in the schema, and is labeled by the label of the edge's target node. An auxiliary node with the root as its only child provides the start state of T. All states of T, except the start state, are final. Let X be the product of the P and T automata. Each schema node and cp/cp+value table is therefore associated to a transition in X.*
   - 2.2. *Navigate the product automaton X and follow the parent-child reference chains in the cp/cp+value tables associated to the transitions.*
3. *When visiting a final state of X via a transition s in the navigation, add to pairList all pairs $(t.child, [t.from, t.to])$ such that t is a tuple in a cp/cp+value table associated to s and t.child is at the end of some parent-child reference chain.*

Parent	Child	From	To	Value
1	4	0	Now	Raptors
2	15	0	Now	Magic
3	25	0	Now	San Antonio

Figure 8: *cp + value* table NBA/franchise/name

4. *When visiting a final state of X via a transition s in the navigation, assign to nodeSet the schema nodes associated to s.*

Algorithm 2 (pathFilter) *Input/Output: nodeSet, pairList. Input: PathExpr, Value.*

1. *Perform steps 1 and 2 from algorithm 1.*
2. *When visiting a final state of X via a transition s in the navigation, add to pairList all pairs (n, I) such that there is a tuple t in a cp+value table associated to s, where t.child is at the end of some reference chain starting at n, and t.value=Value.*
3. *Assign to nodeSet all schema nodes with a cp/cp+value table T such that there is a tuple t in T and a pair $p = (n, I)$ in pairList such that $t.child = n$.*

We now present an example of how a TXPath query can be processed using TempIndex and the document in Figure 1. Consider the query "name of the players playing for the Toronto Raptors, and the corresponding seasons". This is written in TXPath as:

```
//franchise[name='Raptors']//player/name
```

The evaluation process begins following the path NBA/ franchise/name in the temporal schema. From the *cp + value* table of Figure 8, we obtain that node 4, with parent 1, between 0 and *Now* satisfies the condition (note that in Figure 1 the names of the last two teams have been omitted for clarity reasons.) Thus, we must look in the franchise with node 1, and find its players. In the *cp + value* table in Figure 9 we find the team corresponding to franchise in node 1 (node 5). However, according to the temporal schema, we must also look for node 4 in the *cp* table for NBA/franchise/player, depicted in Figure 10 (node 4 is not present in this table). Finally, we join the table NBA/franchise/team, with the tables NBA/franchise/team/player (*cp* table) and NBA/franchise/team/player/name (*cp + value* table), in this order (Figures 11 and 12, respectively), taking into account the time intervals. Note that player 'Carter' is not included in the table of Figure 12, because it is the value of an element *last*, rather than *name*. Also notice that the value for the player in node 7 in Figure 1 (not shown in that figure) is 'Oakley'.

Parent	Child	From	To
1	5	0	Now

Figure 9: *cp* table NBA/franchise/team

Parent	Child	From	To
2	14	0	20
2	16	21	Now
2	24	0	Now

Figure 10: *cp* table NBA/franchise/player

Now, consider the query "name of the players playing in the NBA in 2002". This is written in TXPath as:

`//player[@from` ≤ 2002 `and @to` ≥ 2002`]/name`

Navigating the temporal schema, we find that the players are in two different $cp + value$ tables. The process is analogous to the one described above. However, as there is a temporal condition, we will use the δ_2 and δ_3 tables to obtain the players that were active in 2002.

5.2 Ancestor-Descendant encoding for temporal XML documents

So far we have used *node numbers* for identifying nodes in the XML graph. However, we will show that we can encode nodes in order to improve the performance of some queries in TXPath when using the TempIndex indexing scheme. We devised the *temporal interval encoding*, which is an ancestor-descendant encoding inspired by the interval scheme first presented by Santoro and Khatib [22]. In this scheme, the leaves of a tree are numbered from left to right and each internal node is labeled with a pair of numbers corresponding to its smallest and largest leaf descendants. All known ancestor-descendant encoding schemes (see [13] for a recent survey) are variations of Santoro and Khatib's interval scheme. The average label length of these class of schemes has an upper bound of $2 \log n$, n being the number of nodes in the XML graph. In our index, the integration of the encoding with other index structures allows us to encode the ancestor-descendant relationship using only one number instead of two (the end of each interval is implicitly stored in the order of the δ_k tables).

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

Parent	Child	From	To
5	6	0	20
5	10	0	Now
5	14	23	Now
5	16	0	20

Figure 11: *cp* table NBA/franchise/team/player

Parent	Child	From	To	Value
6	7	0	20	Oakley
14	30	23	Now	Williams
16	17	0	20	McGrady

Figure 12: $cp + value$ table NBA/franchise/team/player/name

In order to formally define the temporal interval encoding, we need to define first a total order relation among the continuous paths in the data graph.

Definition 8 *Let $p_1 = (root, \ldots, v, T_1)$ and $p_2 = (root, \ldots, w, T_2)$ be continuous paths in d. The total order relation $\prec$ is defined as follows:*

1. *If p_i is a proper subpath of p_j, then $p_i \prec p_j$*
2. *Otherwise, let $q_1 = (root, \ldots, n, T_1')$ be the shortest subpath of p_1 that is not a subpath of p_2 and let $q_2 = (root, \ldots, m, T_2')$ be the shortest subpath of p_2 that is not a subpath of p_1.*
   (a) *If $T_1' \bigcap T_2' = \emptyset$ then $p_1 \prec p_2$ iff $T_1'.FROM < T_2'.FROM$.*
   (b) *If $T_1' \bigcap T_2' \neq \emptyset$ then $p_1 \prec p_2$ iff $n < m$, where $<$ is the order relation defined in Proposition 1 in $T_1' \bigcap T_2'$.*

Definition 9 (Temporal Interval Encoding) *Let $\prec$ be the order relation from Definition 8, $succ_\prec$ be the successor function in $\prec$, and* gap *be a function assigning an integer to each continuous path starting at the root. The* temporal interval encoding *function τ is defined over pairs $\langle node, cp \rangle$ by $\tau(n,p) = \tau(m,q) + gap(q)$ if there is a q such that $succ_\prec(q) = p$, $\tau(n,p) = 0$ otherwise.*

Using the temporal interval encoding, Algorithm 1 can be optimized so that it does not need to follow the parent-child reference chains, but instead, given a node-interval pair (m, I), it computes the successor of m, m', and retrieves from the product automaton all the tuples t associated with final state transitions such that $t.child$ is between m and m'.

Example 3 *Consider for instance Figure 13. The* player *node corresponding to 'Williams' has initially been encoded as '67'. This number encodes the node*

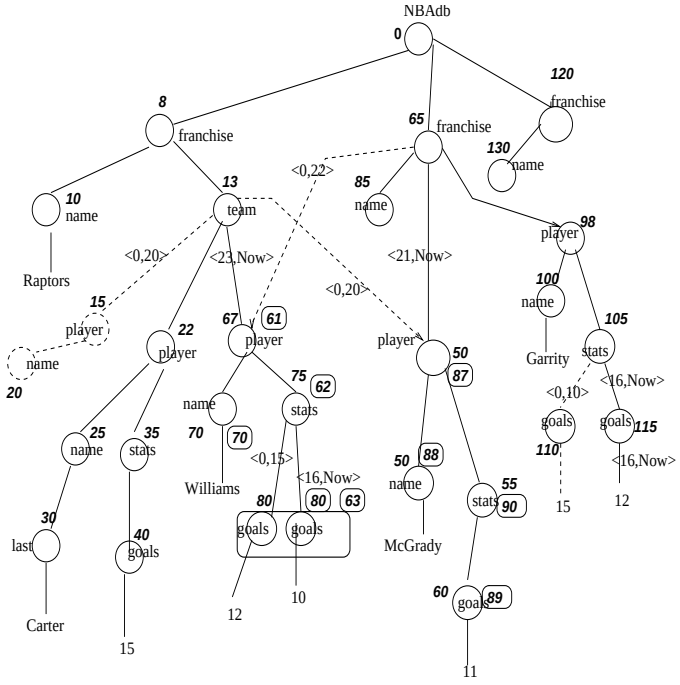

Figure 13: Indexing intervals with *temporal interval encoding*

in the interval $[0, 22]$. *For the interval* $[23, Now]$, *the node's number is '61', because it became a descendant of '13', and must have a number less than '65'. In other words, there are two continuous paths (with disjoint intervals) from the root to the node. For each one of them, we use a different encoding for the same node. Note that these different node numbers do not imply a larger number of tuples in the cp tables, because there is always one tuple for each cp, as in the encoding used so far. For example, a portion of the NBA-franchise-player* cp table *using the encoding of Figure 13 is shown below:*

Child	From	To
87	*21*	*Now*
67	*0*	*22*
61	*23*	*Now*
		...
98	*0*	*Now*

Note that the temporal interval encoding does not require explicitly representing the parent of the target node. Thus, we only keep the attribute 'child' in the cp and cp + value tables.

Now, suppose we are asked for the goals scored by players in the Toronto Raptors. The TXPath expression will be:

```
//franchise[name='Raptors']//goals/text()
```

Answering this query just requires finding node '8' in the NBA-franchise table, checking that the succesor at any interval is node 65, and looking up in the value table for the path NBA/franchise/player/team/stats/goals the nodes with numbers between 8 and 65.

6 Experiments

In this section we will show how indexing temporal intervals and continuous paths improves TXPath query evaluation. We compare TempIndex with two other systems: one index-based and the other DOM-based. We chose ToXin [20] as a representative of the non-temporal XML index class. We choose this particular scheme for convenience, since it is easily available to us; but we believe the results would not be substantially different using any of the other path indexing schemes discussed in Section 2. The second comparison will be against a DOM representation of the base data without any indexing.

Although using a non-temporal index reduces the search space for TXPath queries – compared to the DOM approach – it still does not help with the temporal part of query evaluation. Both ToXin and DOM materialize paths rather than continuous paths; therefore, these two non-temporal backends have to compute the continuous paths involved in a query on-the-fly during query evaluation time. Our experiments will show how important indexing the temporal structure of the data base is for TXPath queries.

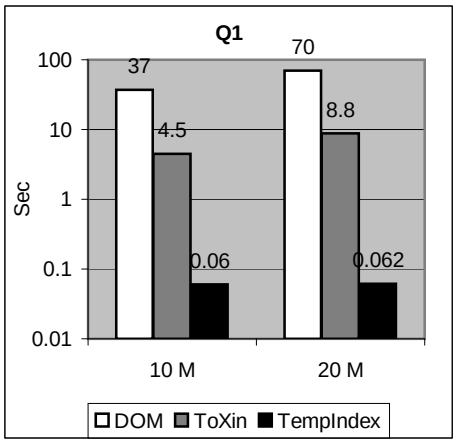

Figure 14: Query Q1 – log scale

For all our experiments we use query processing time as the performance metric. We evaluate the performance of the three systems on a set of five queries, as shown in Table 24. Queries Q1 to Q4 are TXPath retrieval queries, Q5 and Q6 are TXPath update queries, and SN is a document snapshot.

For inserting a node (Query Q5), we specify a time instant t, the new node n' to be inserted, and a current node n (*i.e.* a node with an incoming containment edge where $T_{e_c}.TO = Now$). When deleting a node n at time t_d, (Query Q6) 'Now' is replaced by t_d

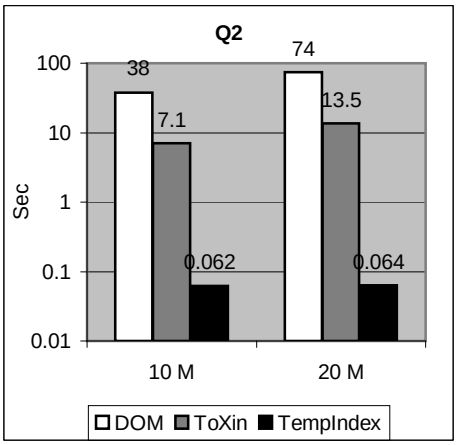

Figure 15: Query Q2 – log scale

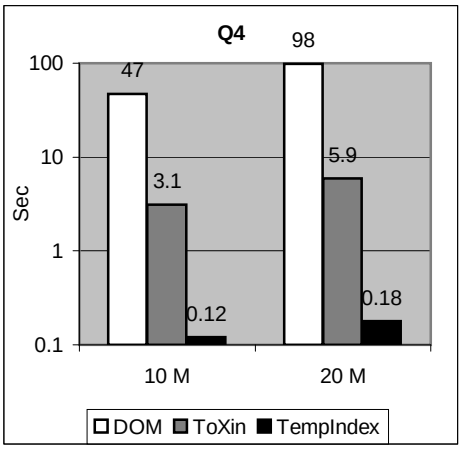

Figure 17: Query Q4 – log scale

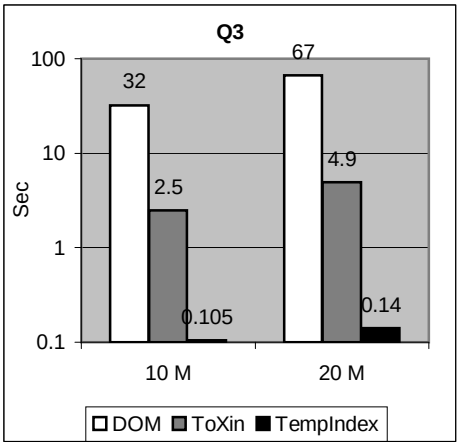

Figure 16: Query Q3 – log scale

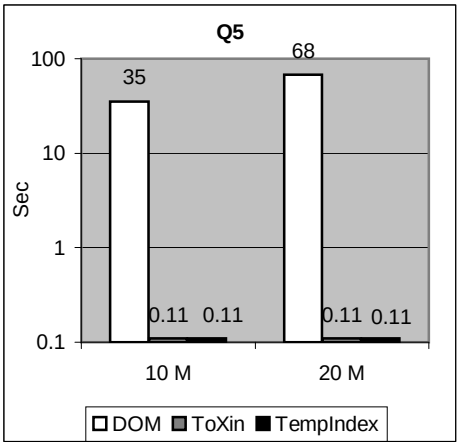

Figure 18: Query Q5: Insert – log scale

in $T_{e_c}.TO$. The same occurs with all the containment edges in the current subtree (the subtree with root n where all the edges e_c have $T_{e_c}.TO = Now$). Reference edges are deleted by setting $T_{e_r}.TO = t_d$ in the temporal label of the edge.

Queries that contain value and interval selections (Q3 through Q6) were performed with ten different combinations of values and intervals. For those four queries we report the average results. We run the queries over the NBA database, which we consider to be a representative example of temporal data. We loaded the data from the NBA web site (www.nba.com) into a relational database (Microsoft SQL Server 2000.) From this database we produced two documents of 10 and 20 Megabytes. We ran all queries over the two documents and the results are reported in Figures 14 to 20. For the experiments we used a Pentium 4 PC at 2Ghz with 1GB of RAM memory and a 60 GB hard drive.

In all retrieval queries TempIndex performed faster than ToXin. The TempIndex speed-up against ToXin ranged from a minimum of nine times (Snapshot–10MB) to a maximum of 210 times (Q2–20MB). Since both systems index label paths and values, the difference in performance can be mostly attributed to the indexing of continuous paths.

Q2 is one of the fastest in TempIndex but one of the slowest in ToXin. The reason for that is that the answer to Q2 is a whole class of continuous paths in the temporal index, which is very easy to find and retrieve using the TempIndex schema. Although in ToXin we can narrow the search by following only those label paths that match the regular expression in the query, we still have to compute all continuous paths over them.

The snapshot, in contrast, requires heavy computation even for TempIndex. We can still narrow the search considerably by using the interval index to locate the classes corresponding to the instant in time we are looking for. However, once these classes are

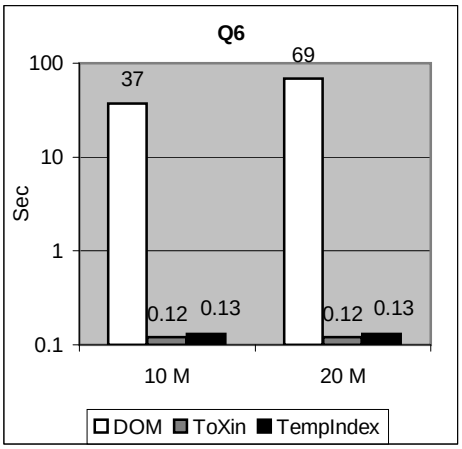

Figure 19: Query Q6: Delete – log scale

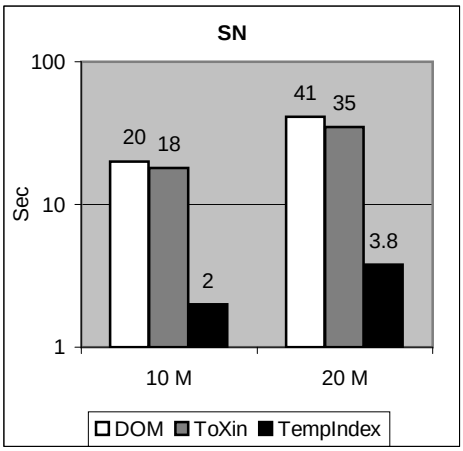

Figure 20: Snapshot – log scale

found we have to reconstruct a whole document navigating back and forth over them. That being said, TempIndex still is almost one order of magnitude faster than ToXin and DOM. Since a path index is not very efficient for document reconstruction operations, the snapshot computation performance of ToXin and DOM are quite similar.

Queries Q1 and Q2 do not contain either interval or value selection predicates and have relatively large answer sets. The answer set of Q1 is closer to the root and smaller than that of Q2. This affects the query processing time in ToXin because the continuous paths to be computed are fewer and much shorter in Q1 than in Q2, with the consequent impact on query evaluation (Q2 takes almost twice the time than Q1). In contrast, since the DOM implementation is not aware of the label path structure of the data graph, it requires the traversal of the whole temporal graph in order to match the regular expression on both Q1 and Q2. Consequently, the difference in query processing

Document size	data graph nodes	*cp's*	temporal index nodes
10 MB	270150	847005	94
20 MB	540300	1694010	94

Figure 21: Data sets and index parameters

Doc. size	TempToxin size	DOM size
10 MB	105 MB	165 MB
20 MB	205 MB	330 MB

Figure 22: Main-memory structure sizes

time between Q1 and Q2 is minimal in DOM.

Queries Q3 and Q4 require the additional computation of value and interval selection, which is reflected in the TempIndex results. In contrast, the size of the answer set and the length of the continuous paths seems to have a bigger impact on ToXin performance than the selection operations, and almost no impact at all in DOM. The reason for that seems to be that ToXin spends most of the query processing time on continuous path computations, while DOM does it on data graph traversal.

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

7 Conclusion and Future Work

We studied the problem of indexing temporal XML documents. We formally described an indexing scheme, denoted TempIndex, composed of three kinds of structures: the temporal schema, the temporal depth tables, and the *cp* and $cp + value$ tables. We showed that materializing *continuous paths* instead of paths increases query performance by several orders of magnitude when compared against index-based and DOM-based implementations of TXPath, the temporal query language that we used. Even snapshots perform one order of magnitude faster, on the average. This performance is due to the fact that the non-temporal backends have to compute on-the-fly the continuous paths involved in a query during query evaluation time.

Our future work includes extending the indexing scheme presented here, in order to support a temporal version of XQuery, and developing a disk-based index for temporal XML documents supporting larger documents.

Doc. size	Answer size				
	Q1	Q2	Q3	Q4	Snapshot
10 MB	3009	7890	450	1300	15400
20 MB	6018	15780	900	2600	30800

Figure 23: Answer sizes of retrieval queries

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

Figure 24: Benchmark queries

Acknowledgments

This work was supported by the Institute for Robotics and Intelligent Systems and the Natural Sciences and Engineering Research Council of Canada.

References

[1] T. Amagasa, M. Yoshikawa, and S. Uemura. A temporal data model for XML documents. In *DEXA*, pages 334–344, 2000.

[2] T. Bozkaya and M. Ozsoyoglu. Indexing valid time intervals. In *DEXA*, pages 541–550, 1998.

[3] S. Chien, V. Tsotras, and C. Zaniolo. Version management of XML documents. In *WebDB*, pages 75–80, Dallas, TX, 2000.

[4] S. Chien, V. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In *VLDB*, pages 291–300, Rome, Italy, 2001.

[5] S. Chien, V. Tsotras, C. Zaniolo, and D. Zhang. Efficient complex query support for multiversion XML documents. In *EDBT*, pages 161–178, Prague, Czech Republic, 2002.

[6] C. Chung, J. Min, and K. Shim. Apex: An adaptive path index for XML data. In *ACM SIGMOD*, pages 121–132, Madison, Wisconsin, 2002.

[7] B. Cooper, N. Sample, M.J. Franklin, G.R Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, pages 341–350, Rome, Italy, 2001.

[8] C.E. Dyreson. Observing transaction-time semantics with TTXPath. In *WISE*, pages 193–202, 2001.

[9] C. Gao and R. Snodgrass. Syntax, semantics and query evaluation in the τXQuery temporal XML query language. *Time Center Technical Report TR-72*, 2003.

[10] C. Gao and R. Snodgrass. Temporal slicing in the evaluation of XML queries. In *VLDB*, pages 632–643, Berlin, Germany, 2003.

[11] M. Gergatsoulis and Y. Stavrakas. Representing changes in XML documents using dimensions. In *XSym*, pages 208–222, Berlin, Germany, 2003.

[12] R. Goldman and J. Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, Athens, Greece, 1997.

[13] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *ACM-SIAM SODA*, pages 954–963, 2002.

[14] R. Kaushik, P. Bohannon, J.F. Naughton, and H. Korth. Covering indexes for branching path queries. In *ACM SIGMOD*, pages 133–144, Wisconsin, Madison, 2002.

[15] R. Kaushik, P. Bohannon, J.F. Naughton, and P. Shenoy. Updates for structure indexes. In *VLDB*, pages 239–250, Hong Kong, China, 2002.

[16] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *IEEE/ICDE*, pages 129–140, San Jose, California, 2002.

[17] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, Rome, Italy, 2001.

[18] T. Milo and D. Dan Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, Jerusalem, Israel, 1999.

[19] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *ACM SIGMOD*, pages 134–144, San Diego, California, USA, 2003.

[20] F. Rizzolo and A.O. Mendelzon. Indexing XML data with ToXin. In *WebDB*, pages 49–54, Santa Barbara, CA, 2001.

[21] B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys, vol. 31,no. 2,pp 158-221*, 1999.

[22] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal (28)*, pages 5–8, 1985.

[23] A. Vaisman, A.O. Mendelzon, E. Molinari, and P. Tome. Temporal XML: Data model, query language and implementation. http://www.cs.toronto.edu/∼avaisman/papers.html, 2004.

[24] P. Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies*, pages 183–202, Philadelphia, 1999.

[25] World Wide Web Consortium. *XML Path Language XPath 2.0*, 2003. http://www.w3.org/TR/2003/WD-xpath20-20030502.