Efficient Handling of Positional Predicates Within XML Query Processing

Zografoula Vagena[1], Nick Koudas[2], Divesh Srivastava[3], and Vassilis J. Tsotras[1]

[1] UC Riverside
{foula,tsotras}@cs.ucr.edu
[2] University of Toronto
koudas@cs.toronto.edu
[3] AT&T Labs-Research
divesh@research.att.com

Abstract. The inherent order within the XML *document-centric* data model is typically exposed through `positional predicates` defined over the XPath navigation axes. Although processing algorithms for each axis have already been proposed, the incorporation of positional predicates in them has received very little attention. In this paper, we present techniques that leverage the power of existing, state of the art methods, to efficiently support positional predicates as well. Our preliminary experimental comparisons with alternative approaches reveal the performance benefits of the proposed techniques.

1 Introduction

XML is gradually becoming the standard for data sharing and information exchange among B2B and other applications over the Internet. Thanks to standard specifications for web services (such as SOAP, WSDL, etc.), user programs can receive requests for data (specified in XML) and return their answers tagged in XML. In addition, via the use of specific query languages such as XPath [6] or XQuery [7], users and applications can compose declarative specifications of their interests, as well as filter and transform data items represented in XML. This widespread acceptance and employment of XML, calls for novel data processing techniques, pertaining to XML's effective storage and retrieval. One key issue in XML query processing is the effective support of the *ordered* model for data and document representation that the language employs. Order is particularly important in the *document-centric* view of the language as it enables exposing the logical structure of the document. Popular XML query languages (e.g., XQuery [7] and XPath [6]) express order sensitive queries through the ability to address parts of a document in a navigational fashion, based on the sequence of nodes in the document tree.

Consider, for example, an XML database of journal articles. For each article entry, the order in which its authors and its sections are listed is relevant. Information extraction tools can take into consideration the tree structure and the positional node order to extract the first two authors of each article, with the following query:

```
//article/child::author[position()<=2]
```

Similarly, the section of an article immediately following the *Related Work* section of the article can be extracted using the query:

```
//article/child::section[./child::title =
    'Related Work']/following-sibling::section[1]
```

Supporting such ordered predicates is thus an important requirement for XML query processing. In the current work we focus on the efficient processing of `positional predicates` within the various XPath navigation axes.

Initial efforts [11, 14] investigated the extent to which relational database technology can support order-based queries. These efforts have concluded that although a relational database performs well in general, in many cases it needs to be extended (i.e. new processing algorithms should be devised) to efficiently support general order-based queries. Although processing techniques that employ direct navigation of the XML tree structure can be trivially adapted to identify nodes that satisfy any positional predicate, recent works have shown that, in the absence of those predicates, such navigational techniques are outperformed by set-based approaches [4, 8, 16]. As a result, it would be beneficial if the latter techniques could be extended to support positional predicates as well.

Set-based solutions transform axes predicates to value conditions and thus reduce the navigation problem to a relational join computation (with the additional requirement of producing the final output in a specific order). Such techniques were originally proposed for queries where navigation is restricted to the `child` or `descendant` axes ([4]). Very recently, set-based methods that take into consideration other navigation axes have appeared ([12, 13, 15]). Moreover, *holistic* solutions (i.e., where multiple steps are grouped and computed as a single operator), have been shown to outperform step-by-step processing, as they avoid unnecessary computations of intermediate results that do not participate in the final answer of the query [8, 9, 15]. All these works introduce new, tree-aware operators to speed up tree traversal, however, none of them considers the existence of positional predicates.

Taking into consideration the abundance and efficiency of processing techniques for the ordered axes, the aim of this paper is to investigate whether such techniques can be modified or extended so as to handle positional predicates as well. In the process, we came to the conclusion that, contrary to common belief, efficient support of positional predicates is non-trivial. Furthermore, not all of the above techniques are applicable when positional predicates are present.

Consider, for example, the `Staircase Join` [12] family of algorithms. Their efficiency is attributed to the fact that while processing queries with navigation, not all context nodes are needed in order to identify the result nodes. Consider the document in Figure 1.a and assume that the `descendant` axis query: `a/descendant::*` is invoked over this document (which finds all descendants of *context* nodes a). Since the descendants of a particular document node x are also descendants of any ancestor of x in the same document, only the context nodes that do not have other context nodes as ancestors are necessary for answering
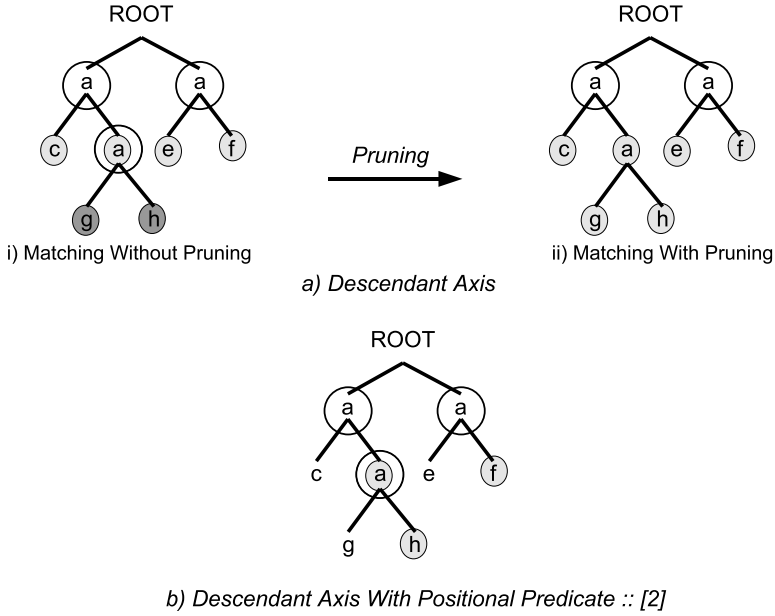
i) Matching Without Pruning   ii) Matching With Pruning

a) Descendant Axis

b) Descendant Axis With Positional Predicate :: [2]

Fig. 1. Inapplicability of Staircase Join in the Presence of Positional Predicates

this `descendant` axis query. In terms of the particular example, only the first and third (in document order) context nodes that are labeled `a` are necessary, while the second `a` can be safely discarded. Similar observations can be made for the other axes as well (e.g. `following`, `following-sibling` etc).

By discarding unnecessary context nodes, one can save matching time from the duplicate results that would otherwise be produced and subsequently be discarded as superfluous. In Figure 1.a(i), if the second `a` node is not discarded, the `g` and `h` nodes are produced twice. Instead, the `Staircase Join` involves a first phase, where it prunes unnecessary context nodes, before performing the actual matching with the appropriate input descendant streams. Figure 1.a(ii) illustrates the matching process after the pruning phase (the result nodes are depicted in gray). Note that for the pruning process, the algorithm only considers: (i) which is the axis involved in the query, and, (ii) the set of context nodes.

Assume now that the positional predicate [position() = 2] is added to the `descendant` axis of the previous example. That is, the query requests only the second descendants under the context nodes. In this case the pruning performed by the `Staircase Join` is not applicable as it would result in false negatives. By discarding the second `a` node in the first phase, one cannot identify the `h` node that belongs to the result, as the second descendant of this `a` node. Preserving only the first `a` node is not enough because `h` is not the second descendant of this `a`. The situation is illustrated in Figure 1.b. While unnecessary context nodes can still be pruned, to identify them, the actual matching over the input streams has to be performed anyways. As a result, the optimization introduced by the `Staircase Join` is inapplicable when positional predicates are present.

In fact, there is a straightforward technique so that positional predicates can be answered as well. In particular, we can: (a) identify all the matching combinations of context and result nodes that satisfy the axis predicate (using any applicable processing algorithm), (b) group on the context nodes and sort each such group on the result nodes, (c) filter from each group the pairs that satisfy the numeric predicate while preserving the result nodes, and, (d) take the union of the result nodes. However this straightforward solution, although acceptable, might lead to unnecessary computation, as it incurs the overhead of identifying context nodes that do not have any matching counterpart that also satisfies the positional predicates. For example, assume the query `a/child::b[4]`, where the fourth `b` child of `a` nodes is requested. If in the document no `a` node has more than three children, then this approach will incur a lot of unnecessary overhead. The question is whether the *"useful"* context nodes can be identified, *prior to* performing all possible matchings. In this paper we will present algorithms that can identify such useful context nodes by effectively maintaining appropriate states of the computation. The contributions of this paper are summarized as:

1. We investigate the problem of supporting queries with positional predicates over XML data. We extend existing state of the art processing algorithms for each of the proposed navigation axes so as to effectively handle positional predicates. To the best of our knowledge, this is the first work that provides a complete, scalable, XML model-aware solution.
2. We target branched queries that may contain *any* number of axes of the same type as well as their backward counterparts (e.g., `descendant` and `ancestor`, or `following-sibling` and `preceding-sibling` etc.)
3. We present a preliminary comparison with (i) the naive solution which checks positional predicates as a post-processing step, (ii) previous techniques that leverage the relational engines to efficiently support tree shaped XML data, and (iii) DOM based XPath query processors. The experimental evaluation reveals the performance advantages of our solutions.

We proceed with Section 2 that provides necessary background while our methods are described in Section 3. In Section 4 we experimentally investigate the performance of the proposed solutions. Finally, Section 5 concludes the paper.

2 Background

Ordered Model. In the data model employed by popular XML query languages [6, 7], a document is represented as an ordered tree, where each node has a unique ID. The *document order* is defined over all nodes and corresponds to the sequence in which nodes occur within the pre-order traversal of the ordered tree [10].

Order Based Predicates. Positional predicates appear in XML query languages in the form of `numeric predicates`. A predicate, filters a sequence of nodes with respect to an axis to produce a new sequence of the nodes for which the predicate evaluates to true. `Numeric predicates` specifically take order into

account and are of the form: `position()` *op* n, where `position()` returns the position of the node within the axis node set, *op* is one of the operators $=$, $<$, $>$, $<=$, $>=$, $! =$ and n is an integer. If a predicate expression evaluates to a number, the result will be true if the number is equal to the context position, otherwise it is false. Thus, a location path `para[3]` is equivalent to `para[position()=3]`. For ease of presentation we describe our techniques in terms of the $=$ operator while pointing out the necessary changes to support the other operators.

Document Representation. The approaches we are going to discuss in this paper project the document into sequences of nodes (we call them `element lists` from now on). There is one sequence per document tag, which maintains all nodes with the same tag. Each node is augmented with information that identifies its position within the XML tree. The position of a node is represented with the triplet: (`left`, `right`, `pright`) where: (a) `left` and `right` are generated by counting tags from the beginning of the document until the start and the end tags of the node are visited, respectively, and, (c) `pright` is the `right` value of its parent node. Using this scheme, the relative positions between any two nodes can be identified in constant time. A node y is a child of a node x if the `right` value of x is equal to the `pright` value of y. Similarly, a node y is `following-sibling` of a node x if the `right` value of x is smaller than the `left` value of y and the `pright` value of x is equal to the `pright` value of y.

3   Algorithmic Approaches

We proceed with the description of our algorithms for supporting XML queries with positional predicates. For the discussion we focus on the `child` and the `following-sibling` axes, and their backward counterparts. The remaining axes can be supported with trivial modifications to the proposed methods. For simplicity we restrict the comparison operator to `equality`. Moreover, while our algorithms follow a set-based evaluation, the results can be easily returned in document order (or any other desired order) providing list-based semantics.

3.1   One-Step Queries

Such queries have the form: `a/child::b`$[n]$ or `a/following-sibling::b`$[n]$, where n is a numerical value. The former query identifies the nth b child of each a document node, while the latter query identifies the nth b sibling of each a document node, which resides after that a in document order.

Child Axis. The optimal processing technique for one-step queries with child axis ('structural joins') was presented in [4]. The input is provided in the form of `element lists` (i.e. the element list of the a nodes and the element list of the b nodes). Each list is sorted on the `left` value of the nodes it contains. The lists for `a, b` are joined in a merge-like fashion. A stack S_a is employed to maintain those `a` nodes that have been visited and whose descendants (and their children) are currently being accessed. At all times during the execution of the algorithm,
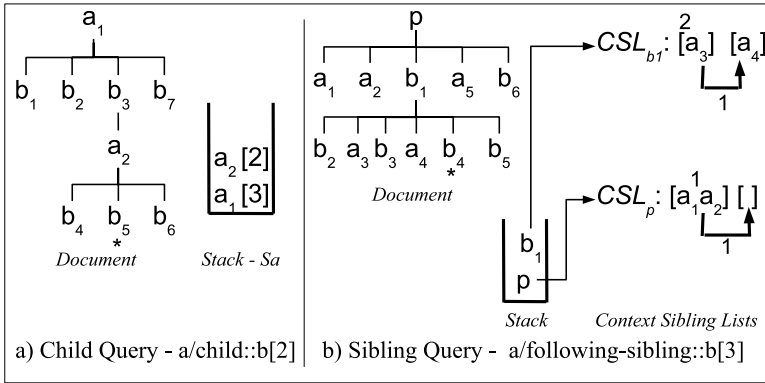
Fig. 2. Processing of positional predicates within one-step queries

each node in S_a is a descendant of the nodes below it in the stack. When a b node is visited, the parent-child relationship, if any, is determined between that node and the current top of S_a. If the accessed b node is not a descendant of the current top of the stack, then it is guaranteed that the top of the stack will not produce further matchings and can thus safely be removed from S_a. The process is then resumed with the new top of the stack.

From the previous discussion it is obvious that in the absence of positional predicates, the relationship between a specific pair of a and b nodes can be determined based solely on their node numberings; the stacks are useful to efficiently identify relationships between sets of a and b nodes. When dealing with positional predicates, however, node numberings do not suffice to determine whether a specific pair of a and b nodes satisfy `a/child::b`[3], for example. Additional state needs to be maintained while processing the sets of a and b nodes to make this inference.

For addressing positional predicates, we note that when a b node is accessed, its a parent (if any) resides in the top of the stack. Moreover, a node a is maintained in S_a until all its children are visited. Since such children are accessed from their element list in document order, the stack provides the appropriate means to keep track of the children nodes for each of the a nodes residing in S_a. In order to achieve that, for each a element already in the stack, we maintain a (child) *counter*. Each time a new b element is accessed and is determined to be child of the current top of the stack, the corresponding counter is incremented. The new value of the counter determines the participation of the b node in the result. In particular, if this value equals to the numerical value n of the positional predicate, the node participates in the result and is joined; otherwise it is discarded. Figure 2.a illustrates the stack state after having accessed node b_5. At this point in the execution, the third child of node a_1 and the second child of node a_2 have been accessed so far.

Following-Sibling Axis. An advantage of the child axis algorithm is that it can perform the join in a *single* pass over the two input element lists. Very

recently, novel merge-like algorithms with the same one-pass property over the input element lists have been independently proposed for the following-sibling axis as well [13, 15]. Here we extend the ideas summarized in [15] so as to support positional predicates as well.

An important observation is that since the input is visited in document order, the following-siblings for some a node occur after all its descendants have been encountered. Equivalently, an a node has to be buffered until the following-siblings of its descendants have been processed first. Moreover, an a node may have other a nodes as following siblings. We call the a nodes that share the same parent as "context-siblings". Context-sibling nodes conceptually form a linked-list, called the `Context-Sibling List`, or CSL in short, which is associated to the parent node of the context-siblings. For example, Figure 2.b shows snapshots of two CSLs, associated with parent nodes p, b_1. New a nodes that are context-siblings are appended at the end of their parent's CSL. CSLs provide an effective way to capture the following-sibling predicate: If a b node becomes a following-sibling to the node at the end of a context-sibling list, it is also a following-sibling to all other nodes currently in that list. There can be many CSLs at a given time (up to the depth of the currently accessed context node in the document tree).

A stack keeps track of the existing CSLs. Each node in the stack is the common parent of the nodes within the corresponding CSL. Moreover it is a descendant of the nodes below it. When a node is removed from the top of the stack (i.e. when all its children have been accessed), its corresponding CSL (if any) is erased. One important point to note here is that although a stack is employed for the processing of both the `child` and the `following-sibling` axis, the contents of the stacks are very different for each algorithm. In particular in the case of the `child` axis, stack entries are from the list of the context nodes, while for the `following-sibling` axis stack entries represent parents of the context nodes. The latter is possible because of the `pright` value that is maintained in the positional representation of each document node.

When the following-sibling query contains a positional predicate, only those following-siblings with position that satisfies the predicate should be joined. Our solution is built on the properties: (i) a context element is accessed before any of its following-siblings, (ii) it is maintained within its CSL, and (iii) its following-siblings are accessed in document order. Therefore, a *counter* could be maintained for each context element in a CSL, that keeps track of the number of following-siblings that have already been identified with respect to this node. In this case whenever a new following-sibling is accessed, the counters of *all* context elements currently within the corresponding CSL would be incremented and only those pairs (if any) that satisfy the numerical predicate would be returned.

Although the previous way of updating the counters is correct, it is not optimal. For example, if there are N a's and b's that are all siblings of each other, then this approach would take time $O(N^2)$ in the worst case for answering `a/following-sibling::b[1]`, even though the output is only $O(N)$. In order to achieve optimality we propose a more complex way of updating the counters, which divides the nodes that reside within a particular CSL into several parti-

tions. We call those partitions `predicate groups` or PGs from now on. Each PG maintains all the nodes whose counters have the same value (i.e. a nodes for which the same number of following-sibling b nodes have been accessed). Moreover each PG is associated with a counter, which is the value of the counter of each node within this PG. The values of those counters from the beginning to the end of a particular CSL list follow a strictly descending order. Furthermore, there exists a pointer from a PG pg_1 to the PG whose counter has the largest value that is smaller than the counter associated with pg_1. Each such pointer is associated with a value, which represents the difference of the values of the counters associated with the two connected PGs. An example of this data structure is represented in Figure 2.b, which captures the state of the algorithm, after node b_4 has been accessed.

Using the above structure, only a constant number of actions have to be performed for counter updates. In particular, when an a node is visited it is appended within the PG that resides at the end of a's associated CSL. When a node b that is associated with a particular CSL is visited, the value of the counter associated with the PG at the beginning of this CSL (we call this PG pg_1) is incremented by one. If no empty PG exists at the end of the CSL, a new empty PG (pg_2) is created at the end of this CSL and the PG that was previously at the end of the CSL is set to point to pg_2, with an associated value of 1. If an empty PG pg_3 exists in the end of the CSL, the value associated with the pointer that points to this pg_3 is incremented by one. Subsequently, if the value of the counter associated with pg_1 does not match the value of the predicate the algorithm resumes with the next document node. Otherwise, the node b is matched with the nodes in pg_1 and the results are returned. At this point the nodes of pg_1 can be discarded as they are not going to create any future results. The counter of the PG referenced by pg_1 is set to the value of the counter of PG minus the associated difference. The following theorem holds for the correctness and performance of processing positional predicates for one step queries (the pseudocode is provided in the appendix):

Theorem 1. *Given a one step query T with an equality positional predicate, `a/axis::b[n]`, where `axis` can be `following-sibling` or `child`, which is invoked over a database D, the proposed processing of the intermediate state for each algorithm correctly identifies all nodes that participate in answers for T in D. Moreover, they have worst-case I/O and CPU time complexities linear to the sum of the sizes of the input lists. Furthermore, the maximum space used is $O(h * f)$, where h is the height of the XML data tree, and f is the maximum fanout at any data tree node.* □

For the one step queries of the theorem, output size is bounded by the sum of the sizes of the input lists. This is not the case for other operators like $position() > 2$. In such cases, our algorithms can be modified to yield correct results with worst-case I/O and CPU time complexities linear to the sum of the sizes of the input lists and the output.

Similar buffering (of context nodes) can be performed for each of the remaining forward axes and hence the functionality of the *counters* can be easily

realized for all forward axes. We thus proceed with our discussion of multi-step queries by focusing on the `following-sibling` axis.

3.2   Multiple Step Queries with *Following-Sibling* Axes

Multiple-step queries involve many query nodes. Each node q_i, may be connected with zero or more other nodes through `following-sibling` axes. We call each q_i that is connected to one or more other nodes a *step-parent* from now on. Such queries can be represented with tree structures whose vertices correspond to query nodes and the edges correspond to the connecting `following-sibling` axes.

One straightforward way to compute a multi-step query with positional predicates, is to divide it into individual steps and use the techniques described in Section 3.1. Nevertheless, this may lead to unnecessary processing of intermediate results. When no positional predicates are present, a more efficient approach is to regard the whole query as a *single* operant and holistically compute the results with a single pass over the input. The proposed algorithm traverses the input in document order and attempts to identify the nodes that satisfy all the predicates of the query at once.
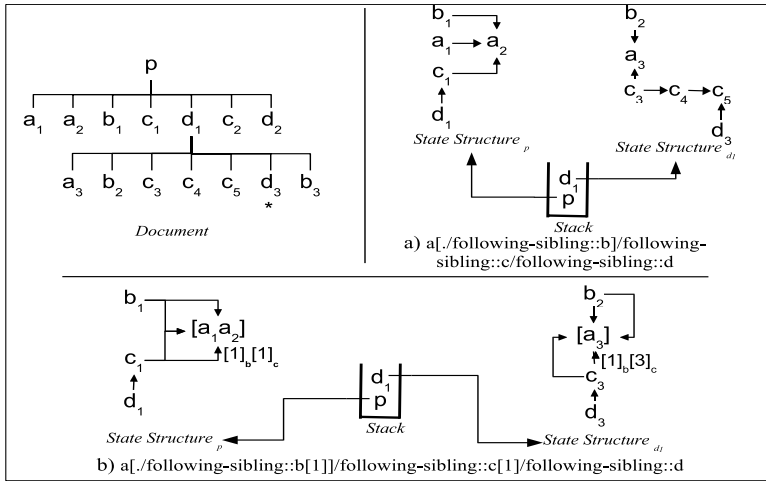

Fig. 3. Processing of positional predicates within multiple-step queries

An important observation that the algorithm utilizes is that any result instance involves only sibling nodes (i.e. nodes that have the same parent). For each such *sibling group* a *state structure* is maintained, which encodes all partial and total results that can be produced from this group with size proportional to the size of the sibling group. The *state structure* consists of one CSL per *step*

parent. The role of a CSL is similar with that in Section 3.1, i.e. to hold context-siblings in document order. An element y within a CSL, maintains a reference (the `step-pointer`) to the latest (in document order) element within the CSL that corresponds to its query parent, with which y is matched. All elements from the beginning of this CSL until the node referenced by the `step-pointer` of y can be matched with y. At each point during the execution of the algorithm multiple *sibling groups* (up to the height of the document tree) can be active (i.e. some but not necessarily all of their nodes have been accessed). A stack is utilized to keep track of all active *sibling groups*. An example of the state structure that is maintained is illustrated in Figure 3.a, where the *state structure* after node d_3 has been accessed is presented.

When positional predicates exist within the multi-step query, counters associated with context nodes within CSLs are also maintained. These counters enable tracking of the following-siblings that have already been seen. PGs are utilized to enable efficient update of those counters in a similar way as described in Section 3.1. The only difference is that when the positional predicate is satisfied, the PG with the matching nodes cannot be discarded until the actual matching takes place. As a result, an additional pointer needs to be maintained pointing to the PG whose counter is updated at each point. To be able to identify the PGs with the matching nodes when total results are produced, the element y maintains two *step-pointers*, one at the first and one at the last PG. All the PGs in between those references contain nodes that can potentially be matched with y. Figure 3.b shows the *state structure* for the multi-step query of Figure 3.a where positional predicates have been added. The computation is depicted right after node d_3 has been accessed.

3.3   Backward Axes

When the multiple-step query contains both *following-sibling* and *preceding-sibling* axes, an adaptation of the method proposed in [5] can convert the query into one with only `following-sibling` axes [15]. The resulting query can be represented with a DAG structure whose vertexes correspond to the query nodes and edges to `following-sibling` axes. In the absence of positional predicates, the additional issue (when compared to the queries in Section 3.2) that needs to be addressed is that a query node may have multiple parents. We call such query nodes as `join` nodes from now on. Our multiple-step algorithm can be easily extended to check those additional predicates. More precisely, a document node y that becomes a `join` node must have at least one corresponding sibling within each of the CSLs of its query parents, before it is inserted within its corresponding CSL, or it triggers the production of results (if y corresponds to a leaf query node).

When considering positional predicates we differentiate whether they are specified on a forward or a backward axis in the original query. If one (or more) of the edges of the DAG that corresponds to a `following-sibling` axis in the original query is associated with a positional predicate, the counters described in

Query: a[./following-sibling::c[1]]/following-sibling::b/preceding-sibling::d[2]
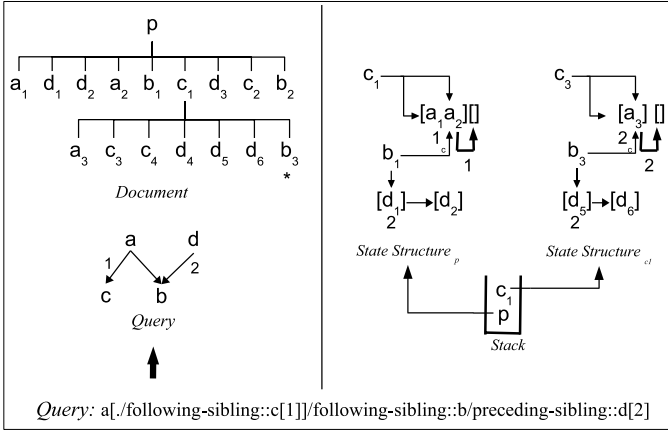
Fig. 4. Integration of positional predicates with backward axes

Section 3.2 provide the necessary means to identify the nodes that satisfy those numeric predicates.

If, on the other hand, the positional predicate is on an edge e that corresponds to a `preceding-sibling` axis in the original query, then the following necessary condition is needed so as a document node y that corresponds to the destination node of e, participates into a result. In particular, there must exist a document node x that corresponds to the source node of e that is the nth preceding sibling of y in *reverse document order*. Nevertheless, such a predicate is easy to check, because our algorithm will have already buffered all the necessary preceding-siblings of y within their associated CSL. As a result, we only need to check whether there exist n elements within this CSL (starting however from the *end* of the list). If such an x element exists, it is the only matching for the y element under consideration. A reference from y to x is then maintained and will be used later in the result production phase. In order to identify those x elements efficiently we utilize the same method as in the case of the forward axes, and we partition the nodes within each CSL into PGs. In the case of the backward axis, however, each PG will contain at most one node and as a result, there is no need to maintain pointers between PGs. Furthermore, in this case it is also unnecessary to maintain the differences between the PG counter values, as those are always equal to 1.

As an example, consider the query in Figure 4, where two positional predicates exist, one on a following-sibling and one on a preceding-sibling axis. The figure also presents the intermediate state after node b_3 has been accessed.

4 Experimental Evaluation

In order to investigate the effectiveness of our techniques, we performed a number of experiments over synthetic, benchmark and real data.
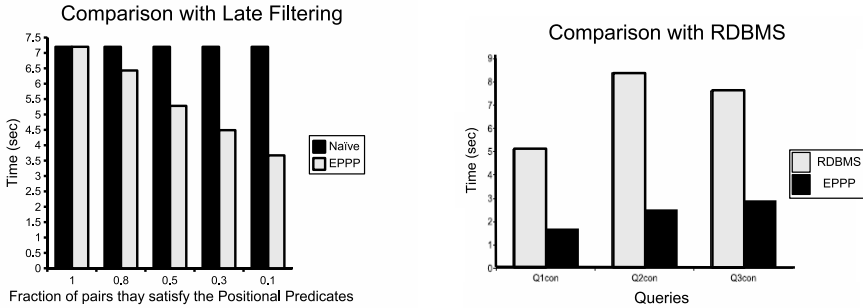
Fig. 5. (a) Comparison with late filtering, (b) Comparison with an RDBMS

4.1 Experimental Setup

We implemented all the algorithms in C++ on top of a native storage manager. All the experiments were conducted on a 2.6 GHz Pentium 4 with 512MB of main memory running RedHat Linux 9. The code was compiled with the GNU compiler version 3.2.2.

To evaluate join performance we measure total execution time for each algorithm. The times with hot cache are reported. We used 8K pages and a 100 block buffer cache.

For the purposes of this section, we refer to our proposed algorithms under the name EPPP (for Effective Positional Predicate Processing). We begin by investigating the importance of early identification of nodes that satisfy the positional predicates. We also compare the performance of EPPP with a relational DBMS. Finally, we investigate the behavior of EPPP with regard to existing main memory XPath processors (Xalan and Saxon, available in [3] and [1]).

4.2 Importance of Early Filtering

This group of experiments investigates whether it is advantageous to identify the context nodes that satisfy the positional predicates *before* performing the matching process. For this experiment we assumed the simple one-step query `a/following-sibling::b`$[position() <= n]$ and generated a large synthetic input dataset (around 100,000 a nodes and around 1 million b nodes). Without any positional predicate the participation of input nodes in the result is 100%.

We first considered the query where no positional predicates exist and compared EPPP with the "positional predicate oblivious" processing algorithm for the following-sibling axes (we call it the `naive` algorithm from now on). Subsequently, we begun changing the value of n, so that at each time only a proportion of the previous results, satisfies the query. For each case we measured the time to completion of each algorithm. For the `naive` algorithm we identify all the results and then filter out the ones that do not satisfy the predicates.

Figure 5.a presents the results. Clearly, when no positional predicates exist, the two algorithms perform very similar. Hence the overhead introduced in

EPPP is very small. This is because the examination of predicate satisfaction has been integrated within the original processing algorithm by introducing simple computations (like counter incrementing or simple boolean expression checking) only among objects that the `naive` algorithm would also access. However, when positional predicates are present, EPPP is becoming more efficient. In particular, the more selective the predicates, the larger the performance gap between the two methods. This is due to the fact that EPPP can discard unnecessary nodes early and before performing the actual matching, while the `naive` algorithm has to incur the overhead of creating all answers, and discard them in a later, post-processing step.

It should be noted that the time differences are expected to be much larger for multiple step queries. This is because the overhead of creating the results is higher (due to the recursive traversal of the CSL through the step pointers). As a result, the performance gain of EPPP will become even more prominent.

4.3   Comparison with an RDBMS

The next set of experiments compares the performance of the EPPP techniques with a commercial relational DBMS, so as to compare against a pure relational approach, that translates the query into SQL. In particular, we followed an approach that combines ideas from [14] and [16]. We first created one table for each of the input streams, where we stored the positional representation of each stream node indexed on `left` and `right`. The dataset used was the 1G (text) database generated by the XMark benchmark, with 825043 `incategory` nodes, 217500 `mailbox` nodes, and 217500 `location` nodes. We used the following-sibling axis queries shown in Table 1(a). Query Q1 is a single-step query, while Q2 and Q3 are multi-step queries.

Table 1. Queries for (a) XMark and (b) real data

Q1 : incategory/following-sibling::mailbox[2]
Q2 : location/following-sibling::incategory/following-sibling::mailbox[2]
Q3 : location[./following-sibling::incategory]/following-sibling::mailbox[2]
Q4 : LINE/following-sibling::STAGEDIR[2]
Q5 : TITLE/following::STAGEDIR[2]
Q6 : PERSONAE[/descendant::PGROUP[2]]/descendant::TITLE

To produce the SQL code, we followed the technique described in [14] and used the $Rank()$ function to produce the results. The relational query was pre-optimized, and in order to decrease the overheads of the logging and recovery subsystems, the query (being the only user query running in the system at the time of the experiment) was evaluated in 'READ UNCOMMITED' access mode.

The results are presented in Figure 5.b; clearly, the relational DBMS approach performs worse than EPPP. As also mentioned in [12, 14], this is because the RDBMS is agnostic to the tree shape of the data and cannot take advantage of it. Moreover, when the number of steps increases, the generated SQL

becomes increasingly complex, which is more prone to optimization errors. Our results confirm the observation that new operators must be added for a relational DBMS to efficiently support XML data retrieval (even more so when positional predicates are present).

4.4   Comparison with XPath Engines

We then compared EPPP with two commonly used XPath engines, namely Xalan [3] and Saxon [1]. Their use is limited to main memory data. For these experiments we utilized real data sets, namely Shakespeare's plays [2]. Each of the documents is small enough to be held in main memory. We evaluated queries Q4, Q5 and Q6 shown in table 1(b) and present the average execution time.

Table 2. Comparison with Xalan and Saxon

	Xalan		Saxon		EPPP	
	with	w/o	with	w/o	with	w/o
Q4	0.20	0.21	0.20	0.12	0.08	0.05
Q5	0.41	0.41	0.18	0.14	0.05	0.07
Q6	0.22	0.21	0.21	0.11	0.04	0.04

The results appear in Table 2. For each query, we report the time with the positional predicate and the time without it. These results show that the EPPP techniques are also efficient when the data is in main memory. These results are very encouraging as they reveal that the EPPP techniques are typically better than specialized XML processors. Moreover, they scale to large datasets and can be easily integrated into general purpose repositories.

4.5   Discussion

We introduced new techniques (EPPP) that provide a general and efficient solution to support positional predicates within navigation axes XPath queries. Our techniques can take advantage of the positional predicates and avoid computation that straightforward approaches, which check the positional predicates as a post-processing step, would entail. An important characteristic of the EPPP techniques is that they identify nodes that participate in the result early in the processing phase of the associated axis. Our preliminary experimental evaluation gives strong evidence that the proposed solutions show more robust performance when compared with pure relational approaches and main-memory specialized XPath engines.

5   Conclusion

We studied the problem of supporting the ordered, tree shaped model of XML data. We proposed efficient methods that extend state of the art processing

techniques for any of the navigation axes, so as the latters can efficiently support positional predicates as well. Most importantly, we showed that the intermediate state (i.e. buffering of nodes) that is maintained by those algorithms provides the necessary means to identify the nodes that satisfy any positional predicate. To the best of our knowledge, this is the first approach that addresses positional predicates in a complete, scalable, XML model-aware fashion.

As future work we intend to investigate query processing techniques that will target heterogeneous queries (i.e. queries where interleaving of any axes is possible). Moreover, we plan to develop methods to support combined efficient evaluation of structural and value-based predicates.

References

1. Saxon xslt and xquery processor. In *Available at http://saxon.sourceforge.net/*.
2. Shakespeare's plays in xml. In *Available at http://www.oasis-open.org/cover/bosakShakespeare200.html*.
3. Xalan xslt processor. In *Available at http://xml.apache.org /xalan-c/index.html*.
4. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, , and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *Proc. of IEEE ICDE*, 2002.
5. C. Barton, P. Charles, M. Fontoura, and V. Josifovski. Streaming xpath processing with forward and backward axes. In *Proc. of ICDE*, 2003.
6. A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Key, J. Robie, and J. Simeon. Xml path language (xpath) 2.0. In *W3C Recommendation. Available from http://www.w3.org/TR/xpath20*, 2005.
7. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language. In *W3C Working Draft. Available from http://www.w3.org/TR/xquery*, 2005.
8. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal xml pattern matching. In *Proc. of ACM SIGMOD*, 2002.
9. T. Chen, J. Lu, and T. W. Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *Proc. of SIGMOD*, 2005 (to appear).
10. M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. Xquery 1.0 and xpath 2.0 data model. In *W3C Working Draft. Available from http://www.w3.org/TR/xpath-datamodel/*, 2005.
11. T. Grust. Accelerating xpath location steps. In *Proc. of ACM SIGMOD*, 2002.
12. T. Grust, M. van Keulen, and J. Teubnem. Staircase join: Teach a relational dbms to watch its (axis) steps. In *Proc. of VLDB*, 2003.
13. G. V. Subramanyam and P. S. Kumar. Efficient handling of sibling axis in xpath. In *Proc. of COMAD*, 2005.
14. I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *Proc. of ACM SIGMOD*, 2002.
15. Z. Vagena, N. Koudas, D. Srivastava, and V. J. Tsotras. Answering order-based queries over xml data. In *Proc. of WWW*, 2005 (poster presentation).
16. C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of ACM SIGMOD*, 2001.

A   Appendix

A.1   Pseudocode for One-Pair Queries

In what follows we provide the pseudocode for one-pair queries of the form:
`a/following-sibling::b`[n]. The algorithm takes as parameters the element
lists that correspond to the a and b query nodes, sorted on the $left$ position of
each node.

Algorithm 1 FSwPosPredicate(EL a, EL b, PREDICATE n)

1: **while** !eof(a) AND !eof(b) **do**
2:    l_a = nextLeft(a) {left position of next node in a list}
3:    l_b = nextLeft(b)
4:    **if** l_a LESSTHAN l_b **then**
5:       n_a = $next(a)$ {retrieve next node from a list}
6:       clearStack(l_a) {also discard CSLs corresponding to popped $pright$ values}
7:       **if** emptyStack() OR (topStack() NOT EQUALS pr_{n_a}) **then**
8:          pushStack(pr_{n_a}) {pr returns $pright$ value of node}
9:       **end if**{append a node to the PG at the end of corresponding CSL}
10:      appendToCSL(n_a, CSL(topStack()))
11:   **else**
12:      n_b = $next(b)$
13:      clearStack(l_b)
14:      **if** !emptyStack() AND (topStack() EQUALS pr_{n_b}) **then**
15:         ++Counter(FIRST-PG(CSL(pr_{n_b})))
16:         **if** NOT-EXISTS-EMPTY-PG-AT-THE-END() **then**
17:            pg = createEmptyPG()
18:            appendPG(CSL(pr_{n_b}), pg)
19:            Difference(pg) = 1
20:         **else**
21:            ++Difference(LAST-PG(CSL(pr_{n_b})))
22:         **end if**
23:         **if** Counter(FIRST-PG(CSL(pr_{n_b}))) == n **then**
24:            OutputMatches() {identify PG referenced by first PG in CSL}
25:            NextPG = NEXT-PG(FIRST-PG(CSL(pr_{n_b})))
26:            Counter(NextPG)=Counter(FIRST-PG(CSL(pr_{n_b})))-
               Difference(NextPG)
27:            DISCARD-PG(FIRST-PG(CSL(pr_{n_b})))
28:         **end if**
29:      **end if**
30:   **end if**
31: **end while**