Bridging the Gap Between Relational and Native XML Storage with Staircase Join

Maurice van Keulen Torsten Grust Jens Teubner

University of Konstanz
Dept. of Computer & Information Science
Box D 188, 78467 Konstanz, Germany
{grust,teubner}@inf.uni-konstanz.de

University of Twente
Faculty of EEMCS
Box 217, 7500 AE Enschede, The Netherlands
m.vankeulen@utwente.nl

March 4, 2003

Abstract

Several mapping schemes have recently been proposed to store XML data in relational tables. Relational database systems are readily available and can handle vast amounts of data very efficiently, taking advantage of properties that are specific to the relational model, like sortedness or uniqueness. Tables that originate from XML documents, however, carry some further properties that cannot be exploited by current relational query processors. We propose a new join algorithm that is specifically designed to operate on XML data mapped to relational tables. The *staircase join* is fully aware of the underlying tree properties and allows for I/O and cache optimal query execution. As a local change to the database kernel, it can easily be plugged into any relational database and allows for various optimization strategies, e. g. selection pushdown. Experiments with our prototype, based on the *Monet* database kernel, have confirmed these statements.

1 Introduction

There's no doubt about the important role XML will play in tomorrow's database systems. The key issue from a database point of view is XML's data model, namely the tree. Current database technology can handle various kinds of relational data very well. For efficient XML processing, however, they still lack an awareness of the *tree* structure.

Several proposals try to bridge this gap by mapping the XML tree structure into relational tables [2] and use existing relational databases to store them. Although these approaches can benefit from the advanced indexing techniques of the RDBMS, the database kernel does not know the actual (tree) origin of the data and hence cannot profit from this information. It can only rely on its own statistics in the search for efficient query plans.

The *staircase join* is a new join algorithm that can easily extend an existing relational database. It is tailored for our *XPath accelerator* mapping scheme presented in [3] and makes the database kernel fully aware of the underlying tree structure.

The staircase join supports the performance-critical XPath axes `descendant`, `ancestor`, `following`, and `preceding`. For arbitrary context *sets*, our algorithm returns a duplicate-free, sorted sequence of result nodes, as required by the XPath specification [1].

This paper will first give a brief overview of the XPath accelerator mapping scheme in Section 2 and point out its relevant properties. These properties will lead to the customized staircase join algorithm that will be described and refined in Section 3. The experimental results in Section 4 will prove the efficiency of our algorithm before Section 5 gives a short summary of the paper.

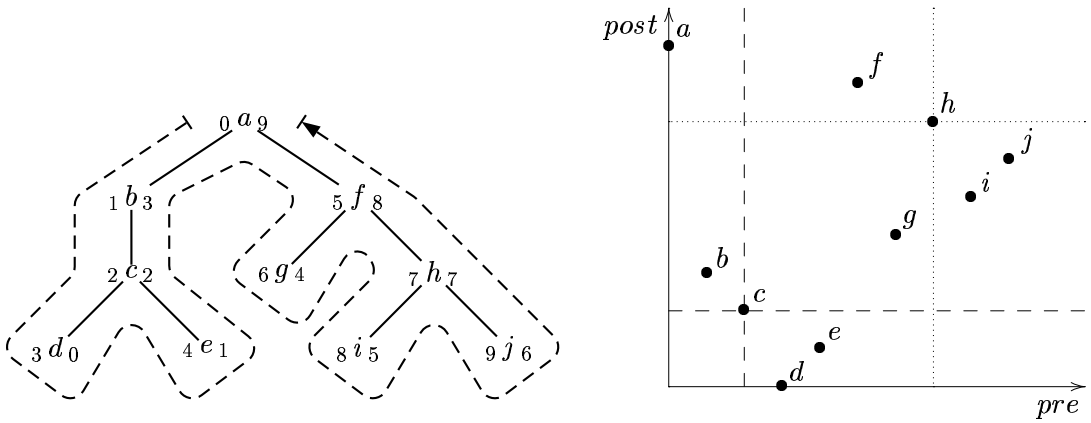

Figure 1: Determination of the values $pre(v)$ (left-hand numbers) and $post(v)$ (right-hand numbers) in a single tree walk, and node distribution in the *pre/post* plane. For each node, the four major XPath axes partition the whole document into four regions, depicted for the nodes c and h with dashed and dotted lines, respectively.

2 The XPath Accelerator Mapping Scheme

Throughout this paper we will treat an XML document as a tree solely consisting of element nodes.[1] The XPath Accelerator mapping scheme assigns to each node in the XML tree a pair of integer values that fully describes the structure of the document:

(a) The *preorder rank* $pre(v)$ is the node order for a *preorder traversal* of the whole tree, i. e. a tree node v is visited *before* its children are recursively traversed from left to right.

(b) The *postorder rank* $post(v)$ is the node order for a *postorder traversal*, i. e. a node v is visited *after* all its children have been traversed from left to right.

An example document tree and its *pre* and *post* values are given in Figure 1. Observe that the *pre*-order is exactly XPath's document order.

With this numbering scheme, the result set of the four XPath axes `descendant`, `ancestor`, `following` and `preceding` can be described with simple range conditions on the *pre/post* values. Due to their recursive definition, these axes are usually hardest to implement.

$$v' \text{ is a descendant of } v \quad \Leftrightarrow \quad pre(v') > pre(v) \wedge post(v') < post(v) \tag{1}$$

$$v' \text{ is an ancestor of } v \quad \Leftrightarrow \quad pre(v') < pre(v) \wedge post(v') > post(v) \tag{2}$$

$$v' \text{ is a following node of } v \quad \Leftrightarrow \quad pre(v') > pre(v) \wedge post(v') > post(v) \tag{3}$$

$$v' \text{ is a preceding node of } v \quad \Leftrightarrow \quad pre(v') < pre(v) \wedge post(v') < post(v) \tag{4}$$

In a *pre/post plane*, these conditions are represented in an illustrative way. Each node induces a partitioning of the whole document into four regions, that correspond to the result sets of the above axes. For the remainder of this paper, we will therefore call these axes *major axes*. The result sets are the nodes in the lower-right, upper-left, upper-right and lower-left partition of node v, respectively. Figure 1 depicts this partitioning for an example tree.

With conditions (1-4) it is straightforward to map XPath queries to SQL. In [3] we described, how any XPath query can be mapped to a single SQL query, e. g.:

e`/descendant::node()` =

```
SELECT DISTINCT p.*
  FROM e v, accel p
 WHERE p.pre > v.pre AND p.post < v.post
 ORDER BY p.pre
```

(5)

[1] The model can easily be extended to cover attributes, text nodes, processing instructions, etc. as well.

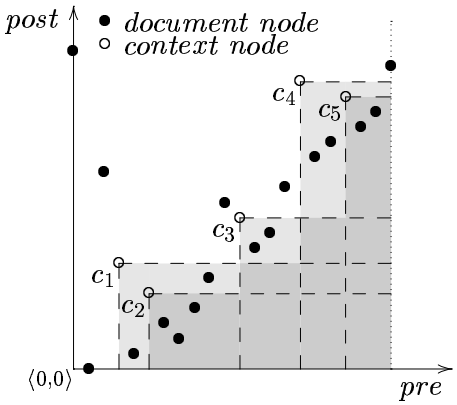

Figure 2: Overlapping regions in the *pre/post* plane (context nodes c_i).

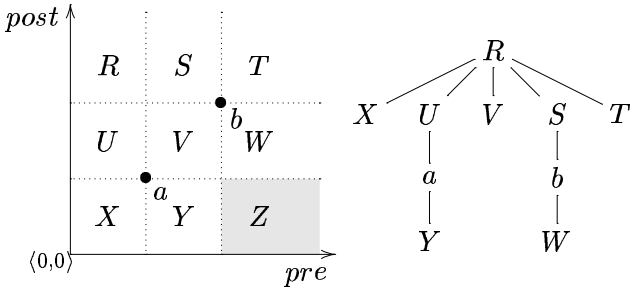

Figure 3: Empty region in the *pre/post* plane and the corresponding tree structure. Nodes that are in a preceding/following relationship cannot have common descendants, and the shaded region must be empty.

3 Efficient Implementation: The Staircase Join

Off-the-shelf RDBMS will use an R-tree index (if available) or a combination of two B+-tree indices to support these region queries. In our experiments described in [4], IBM's DB2 V7.1 used two B+ indices and an index intersection to answer queries like example (5). Note, however, that we need the expensive `DISTINCT` and `ORDER BY` operators to ensure XPath semantics.

Both indices assume a random node distribution in the *pre/post* plane. Although we actually know much more about the node distribution due to its tree-origin, the database cannot profit from this knowledge for query execution.

With a closer look at our mapping scheme, we can derive a number of important implications that will help the staircase join to avoid unnecessary work and optimize its processing. For the sake of brevity, we will focus on the `descendant` axis for the remainder of this paper; the other major axes can be optimized similarly.

3.1 Redundant Context Nodes

According to the XPath semantics, the result of any location step is a *duplicate free* ordered sequence. Context nodes that lie within the result region of any other context node will therefore not contribute any new nodes to the result sequence and can easily be removed from the context set. A *pruning* phase removes these redundant nodes and reduces work for the actual staircase join.[2]

Figure 2 depicts the efficiency of pruning. The dark shaded regions are in the result set of a `descendant` step for more than one context node. Pruning the redundant context nodes c_2 and c_5 reduces the overlap.

3.2 Empty Regions in the *pre/post* Plane

The example in Figure 2 shows that context pruning does not completely eliminate the region overlap. Taking into account the tree-origin of the node distribution in the *pre/post* plane, it is easy to see that the remaining overlapping regions are actually all empty.

This becomes obvious in Figure 3 that illustrates the situation for two context nodes a and b schematically. Two context nodes that are in a preceding/following relationship can never have

[2] Pruning is described as a separate execution phase here. In our actual staircase join implementation, however, we combined pruning and execution into a single phase, leading to no overhead from pruning.

common descendants, and so the shaded region must be empty. This is an immediate result from the fact that what we store is actually a tree.

An important implication is that we won't encounter duplicate result nodes after context set pruning. But the knowledge about empty regions also helps staircase join to avoid unnecessary work. Each point in the remaining scan area can be reduced to exactly one context node and even for large context sets we need to compare each point in the *pre*/*post* plane with exactly one context node.

3.3 The Staircase Join Algorithm

In particular that last implication is reflected in the basic staircase join algorithm, shown as Algorithm 1. For each node in the context set, staircase join scans the corresponding interval in the *pre*/*post* plane along the *pre* axis and collects its result nodes. (Note that we assume the context set and the doc table be sorted in *pre*-order. The context set needs to be pruned.)

Algorithm 1: Basic staircase join algorithm

```
 1 staircasejoin_desc
   (doc : TABLE (pre,post),
    context : TABLE (pre,post)) ≡
 2 BEGIN
 3   result ← NEW TABLE (pre, post);
 4   FOREACH SUCCESSIVE PAIR (c1, c2) IN context DO
 5     FOREACH NODE n IN doc
 6       FROM n.pre > c1.pre TO n.pre < c2.pre DO
 7         IF n.post < c1.post THEN
 8           APPEND n TO result;
 9   c1 ← LAST CONTEXT NODE IN context;
10   FOREACH NODE n IN doc WITH n.pre > c1.pre DO
11     IF n.post < c1.post THEN
12       APPEND n TO result;
13   RETURN result;
   END
```

The algorithm is perhaps most closely described as a merge join with a dynamic range predicate. Document table and context set are both scanned sequentially and only once. Despite from beeing very cache-friendly, this has an important impact on the result set for XPath step evaluation. Staircase join will never deliver duplicates and the result will always be sorted in document order.

3.4 More Tree-Aware Optimizations

So far we have not fully exploited the observation about empty regions in Section 3.2. This tree-specific property of the *pre*/*post* plane can be used to *skip* parts of the document table and avoid unnecessary scans.

Each time we miss the condition in lines 7 and 11, we have a situation as depicted in Figure 3 with context node c_1 as a and document node n as b. So the remainder of the currently scanned interval must be a region like region Z in Figure 3 and therefore be empty. We can safely skip forward to the next scanning interval and modify the IF clauses in Algorithm 1 to

```
IF n.post < c1.post THEN
  APPEND n TO result;
ELSE
  BREAK;
```

The effectiveness of this skipping is high. Each node visited either contributes to the result or leads to a skip. We will therefore never touch more than |context| + |result| nodes in the *pre*/*post* plane, while the basic Algorithm 1 would scan along the entire plane, starting from the context node with minimum preorder rank.

4 Experimental Results

5 Summary

```
staircasejoin_desc (doc : TABLE (pre,post),
                    context : TABLE (pre,post)) ≡
BEGIN
  result ← NEW TABLE (pre, post);
  FOREACH SUCCESSIVE PAIR (c1, c2) IN context DO
[5]   scanpartition (c1.pre + 1, c2.pre − 1, c1.post);
  c ← LAST CONTEXT NODE IN context;
  n ← LAST DOCUMENT NODE IN doc;
  scanpartition (c.pre + 1, n.pre, c.post);
  RETURN result;
END
```

```
scanpartition (pre1, pre2, post) ≡
BEGIN
  FOR i FROM pre1 TO pre2 DO
    IF doc[i].post < post THEN
      APPEND doc[i] TO result;
END
```

Algorithm 2: foo

References

[1] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML path language (XPath) 2.0. Technical Report W3C Working Draft, Version 2.0, World Wide Web Consortium, 11 2002.

[2] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report, INRIA, Rocquencourt, 1999.

[3] Torsten Grust. Accelerating XPath location steps. In *Proc. of the 21st Int'l ACM SIGMOD Conference on Management of Data*, pages 109–120, Madison, Wisconsin, USA, 6 2002. ACM Press.

[4] Torsten Grust, Maurice van Keulen, and Jens Teubner. On accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems*, 2003 (under revision).

[5] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.