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

Jens Teubner° Maurice van Keulen• Torsten Grust°

°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

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 physical 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 [3] 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* operator is a new join algorithm that can easily extend existing relational databases. It is tailor-made for our *XPath accelerator* mapping scheme presented in [4] and makes the database kernel fully aware of the underlying tree structure.

The Staircase join supports the most 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.

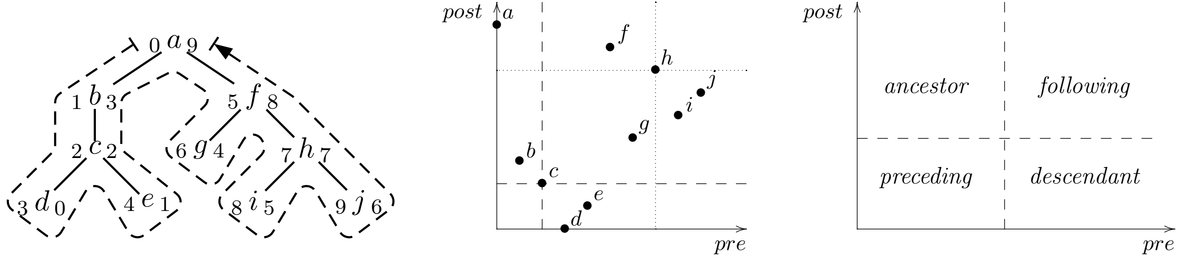


Figure 1: Tree walk to determine the values *pre*(v) (left-hand numbers) and *post*(v) (right-hand numbers), and node distribution in the *pre*/*post* plane. For each node, the four major XPath axes partition the whole document into four regions, depicted for the nodes c and h with dashed and dotted 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 describe 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 corresponds exactly the document order defined by the XML specification.

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 efficiently.

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

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

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

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

In a *pre/post plane* (Figure 1), 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. 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 [4] we described how any XPath query can be mapped to a single SQL query, e. g.:

```
                               SELECT DISTINCT p.*
                                FROM e c, doc n
e/descendant::node()  =        WHERE n.pre > c.pre AND n.post < c.post      (5)
                               ORDER BY n.pre
```

3 Efficient Implementation: The Staircase Join

Off-the-shelf RDBMSs 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 [5], IBM's DB2 V7.1 used

[1] The extension to attributes, text nodes, processing instructions, etc. is straightforward.

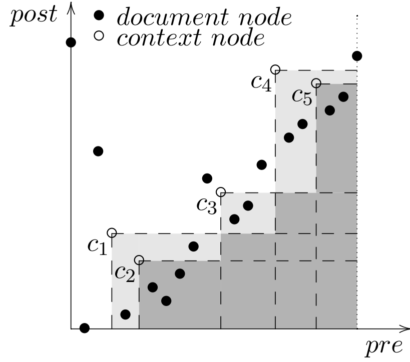


Figure 2: Overlapping regions in the *pre*/*post* plane (context nodes c_i).

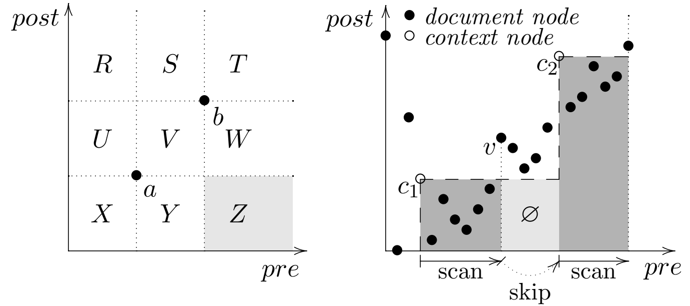


Figure 3: Empty region in the *pre*/*post* plane. Knowledge about empty regions makes it possible to *skip* parts of the document.

two B+ indices and an index intersection to answer queries as in 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 from 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

Following 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 new nodes to the result sequence and can just 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 effectiveness of pruning. The darkly 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. But taking into account the tree origin of the node distribution in the *pre*/*post* plane, it is easy to see that the remaining overlap regions are actually all empty.

This becomes obvious in Figure 3 (left) that schematically illustrates the situation for two context nodes a and b. Two context nodes that are in a preceding/following relationship can never have common descendants, and so the shaded region must be empty. This is an immediate result from the fact that the stored data actually is 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 the 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 node in the context set.

---

[2]Pruning is described as a separate execution phase here. In our actual staircase join implementation, however, we merged pruning and execution into a single phase, avoiding overhead from pruning.

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, the 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 might be described best as a merge join with a dynamic range predicate. Document table and context set are both scanned sequentially and only once. Despite being very cache-friendly, this has an important impact on the result set for XPath step evaluation. The 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 empty regions observation from Section 3.2. This tree-specific property can be used to *skip* parts of the document table and avoid unnecessary scans.

Each time the condition in lines 7 and 11 fails, 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 jump 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;   /* skip */
```

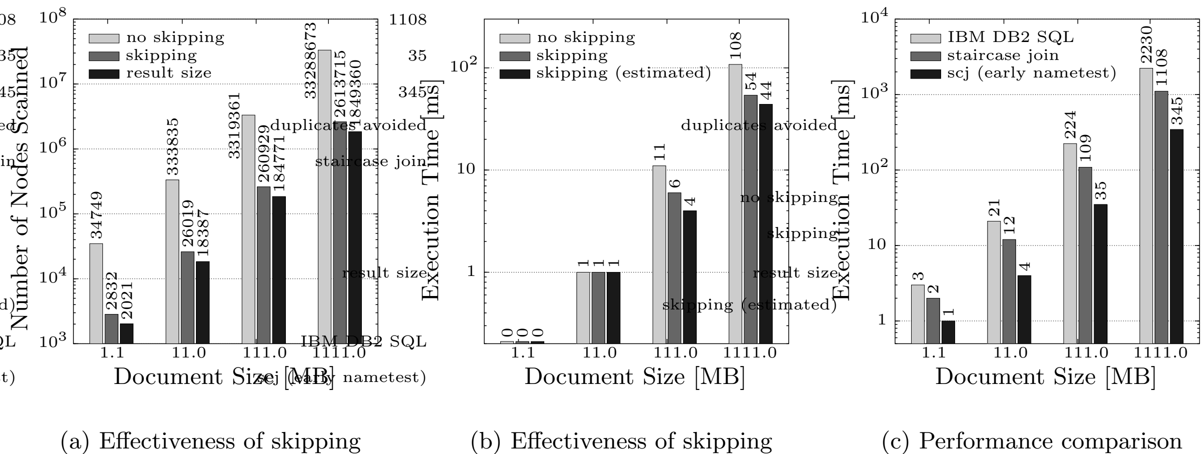
The effectiveness of this skipping is high. Each node visited either contributes to the result or leads to a skip. We will therefore never touch more than |context| + |result| nodes in the *pre*/*post* plane, while the basic Algorithm 1 would scan along the entire plane, starting from the context node with minimum preorder rank. Figure 3 (right) illustrates skipping.

4 Experimental Results

To validate the above ideas we implemented a staircase join operator for the Monet database kernel [2]. The results in Figure 4 have been established on a 2.2 GHz Pentium 4 machine, equipped with 2 GB RAM and running Linux kernel 2.4. We used XML documents generated by the XML generator *XMLgen* from the XMark benchmark project [6] and executed the XPath query `/descendant::profile/descendant::education`[3].

Figures 4(a) and 4(b) demonstrate the effectiveness of the skipping optimization. The number of nodes processed in the second step of our query is reduced significantly. About 90 % of the irrelevant nodes have been skipped. Figure 4(b) shows that execution time for the second step is roughly cut by half. The third algorithm depicted uses a slight modification of the skipping

[3]The first provides the context set for staircase join, while the second step is the actually "hard" step.



(a) Effectiveness of skipping

(b) Effectiveness of skipping

(c) Performance comparison

Figure 4: Experimental results (All diagrams use a logarithmic scale.)

technique that is optimized for main-memory processing ("skipping (estimated)").

In Figure 4(c) we compare the total query execution times to the commercial database system IBM DB2 UDB 7.1. The middle set of bars shows the results for the staircase join with a subsequent name test. In our introduction, we already proposed that known query rewriting techniques can be applied to the staircase join as well. The third set of bars shows how execution time can be reduced by a factor of about 3 when the name test is done *before* the evaluation of the staircase join. We believe that this flexibility makes the staircase join a very promising approach to use relational database technology for highly efficient XML query processing.

5 Summary

The staircase join operator incorporates detailed knowledge about the properties of the *pre*/*post* plane. It uses tree-specific properties like ancestor-descendant relationships for efficient XPath processing. We propose that the staircase join can be added to existing relational database systems. Known optimization and query rewriting techniques like selection pushdown can be applied to the staircase join, which makes staircase join-extended databases a very promising platform for XML storage and query processing.

References

[1] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M. F., Kay, M., Robie, J., and Siméon, J. XML Path Language (XPath) 2.0. Tech. Rep. W3C Working Draft, Version 2.0, World Wide Web Consortium, November 2002.

[2] Boncz, P. A. *Monet — A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, University of Amsterdam, The Netherlands, May 2002.

[3] Florescu, D., and Kossmann, D. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Tech. rep., INRIA, Rocquencourt, France, 1999.

[4] Grust, T. Accelerating XPath Location Steps. In *Proc. of the 21st Int'l ACM SIGMOD Conference on Management of Data* (Madison, Wisconsin, USA, June 2002), ACM Press, pp. 109–120.

[5] Grust, T., van Keulen, M., and Teubner, J. On Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems* (2003 (under revision)).

[6] Schmidt, A. R., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., and Busse, R. XMark: A Benchmark for XML Data Management. In *Proc. of the International Conference on Very Large Data Bases (VLDB)* (Hong Kong, China, August 2002), pp. 974–985.