An Injection with Tree Awareness: Adding Staircase Join to PostgreSQL

Sabine Mayer° Torsten Grust° Maurice van Keulen• Jens Teubner°

°University of Konstanz
Department of Computer and Information Science
P.O. Box D 188, 78457 Konstanz, Germany
{mayers,grust,teubner}@inf.uni-konstanz.de

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

1 Introduction

The syntactic wellformedness constraints of XML (opening and closing tags nest properly) imply that XML processors face the challenge to efficiently handle data that takes the shape of *ordered, unranked trees*.

Although RDBMSs have originally been designed to manage table-shaped data, we propose their use as XML and XPath processors. In our setup, the database system employs a relational XML document encoding, the *XPath accelerator* [1], which maps information about the XML node hierarchy to a table, thus making it possible to evaluate XPath expressions on SQL hosts.

Conventional RDBMSs, nevertheless, remain ignorant of many interesting properties of the encoded tree data, and were thus found to make no or poor use of these properties. This is why we devised a new join algorithm, the *staircase join* [2], which incorporates the tree-specific knowledge required for an efficient SQL-based evaluation of XPath expressions.

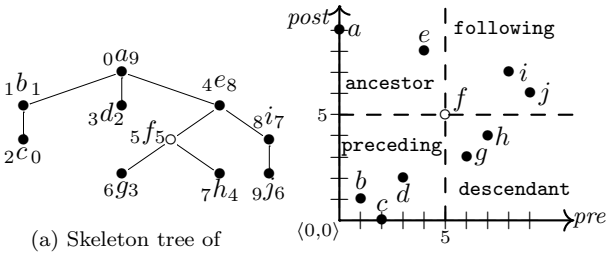
In a sense, this demonstration delivers the promise we have made at VLDB 2003 [2]: a notion of tree awareness can be injected into a conventional disk-based RDBMS kernel in terms of staircase join. The demonstration features a side-by-side comparison of both, an original and a staircase-join enhanced instance of PostgreSQL [4]. The required changes to PostgreSQL were local, the achieved effect, however, is significant: the demonstration proves that this injection of tree awareness turns PostgreSQL into a high-performance XML processor that closely adheres to the XPath semantics.

2 Staircase Join

2.1 XPath Accelerator and Pre/Post Plane

The *XPath accelerator* [1] encodes the tree structure of an XML document using unique pairs of integer values, the nodes' *preorder* and *postorder traversal ranks*.

If these ranks are used to place the document nodes in the two-dimensional *pre/post plane* (Figure 1), it becomes apparent that the encoding preserves an important property. Any context node v divides the XML document into four disjoint regions, whose union plus v itself covers all nodes of the document. The four regions correspond to the result of the XPath location steps v/`preceding`, v/`ancestor`, v/`following`, and v/`descendant`, respectively.[1]



(a) Skeleton tree of XML document and pre/postorder ranks.

(b) Resulting pre/post plane.

Figure 1: The regions associated with the four major XPath axes in the pre/post plane. Context node is f.

The nodes of the plane are maintained in a table $doc = \underline{pre \mid post}$, the *document table*. The document nodes n contained in the respective plane regions may then be defined for any arbitrary context node $v \in doc$ by simple conjunctive range queries:

$$
\begin{aligned}
n \in v\texttt{/preceding} &\Leftrightarrow v.pre > n.pre \wedge v.post > n.post \\
n \in v\texttt{/following} &\Leftrightarrow v.pre < n.pre \wedge v.post < n.post \\
n \in v\texttt{/descendant} &\Leftrightarrow v.pre < n.pre \wedge v.post > n.post \\
n \in v\texttt{/ancestor} &\Leftrightarrow v.pre > n.pre \wedge v.post < n.post
\end{aligned}
$$

[1] These four axes constitute the focus of our demonstration. We will refer to them as *major XPath axes* in the following. For the treatment of further XPath features, *e.g.*, node and name tests, please refer to [1, 2].

These region queries enable us to translate XPath path expressions into SQL queries. Each location step in a given expression is converted into a join which links the initial context node set *context* or the result of the previous location step to the document table. The join predicates directly correspond to the region queries. Thus, the XPath expression Q_1 = *context*/`following`/`descendant` will be translated into the following SQL query:

```
SELECT DISTINCT n2.*
  FROM context v, doc n1, doc n2
 WHERE v.pre < n1.pre AND v.post < n1.post
   AND n1.pre < n2.pre AND n1.post > n2.post
 ORDER BY n2.pre;
```

The `DISTINCT` and `ORDER BY` clauses make the result comply with the W3C XPath semantics: nodes are returned in document order with duplicates removed.

2.2 Pruning, Partitioning, and Skipping

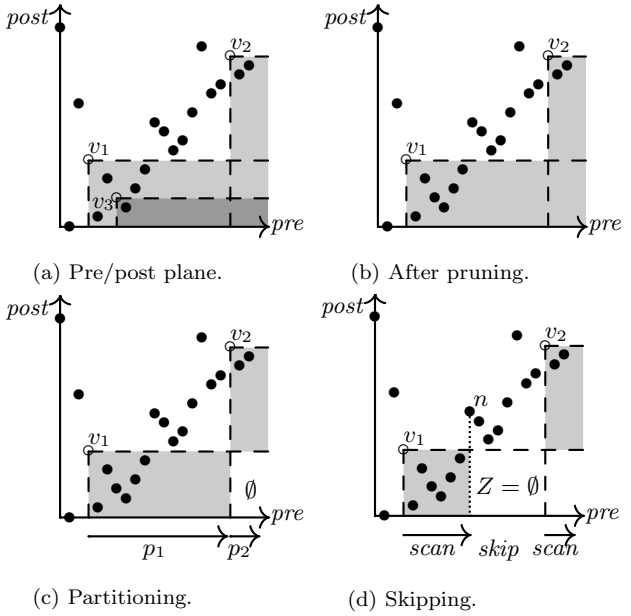
In a conventional RDBMS, this evaluation of an XPath location step amounts to query plans in which the computation of the region queries happens on a per-context-node basis, *i.e.*, it will typically involve several rescans of the document table.

In contrast to that, *staircase join* [2] employs three techniques (*pruning*, *partitioning*, and *skipping*) which devise a significantly more efficient way to work with tree-structured data. Most importantly, staircase join makes sure that the evaluation of an XPath location step requires at most one sequential scan of the document table and that the result of each location step is duplicate-free and sorted in document order.

Context pruning reduces the work load by removing redundant nodes from the context set. Figures 2 (a) and (b) show how pruning works for the `descendant` axis. The removal of nodes is based on *inclusion*, which means that the `descendant` region of context node v_3 is completely contained in the `descendant` region of v_1. For the `preceding` and `following` axes, pruning even reduces the context set to a single node.

Partitioning ensures that one sequential scan of the document table is enough to evaluate an XPath axis. Since the node distribution in the pre/post plane is isomorphic to the XML tree structure, certain plane regions are guaranteed to not contain any nodes ($\emptyset$ in Figure 2 (c)). Staircase join uses this observation to avoid unnecessary rescans of the plane.

Skipping reduces the number of document nodes that must be considered during the evaluation of a partition. Figure 2 (d) shows an example of `descendant` axis skipping. As soon as we come across the first `following` node n of context node v_1, we know, again due to the tree isomorphism, that region Z is necessarily empty and the remaining nodes in the partition may be skipped.



(a) Pre/post plane. (b) After pruning. (c) Partitioning. (d) Skipping.

Figure 2: The pre/post plane before (a) and after pruning (b), partitioning (c), and skipping (d) for a `descendant` location step. Context nodes v_1, v_2, v_3.

3 Tree Awareness for PostgreSQL

Completely encapsulated inside staircase join, we injected this awareness of the XML tree structure into PostgreSQL 7.3.3 [4]. The integration mainly affected two query processing stages [3].

3.1 Planning/Optimization

During planning/optimization, we detect the cases in which staircase join is the optimal join method. The decision is based on an examination of the join clauses (region queries): (1) both operands of a *staircase join clause* must be of data type `tree`[2], (2) there must be two such clauses (the *pre* and *post* clause), and (3) their comparison operator combination must specify a valid XPath axis (*e.g.* (<,<) for the `following` axis).

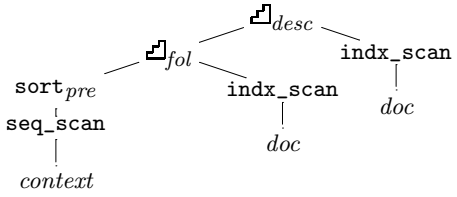


Figure 3: Execution plan for query Q_1 of Section 2.1.

The typical execution plan of an SQL-based XPath query is shown in Figure 3. As the staircase join (⛍)

[2]The data type was newly introduced into PostgreSQL to indicate that a column contains tree-structured data. It is a derivative of the SQL `int` type.

produces identical execution cost for both types of linear join trees, only the left-deep variant is considered, *i.e.*, the current context set will always be the left (outer) input parameter of the join and the document table the right (inner).

3.2 Execution

Staircase join was adapted to fit into PostgreSQL's execution environment. This involved a local change to the executor, *i.e.*, the introduction of a new execution module which implements pruning, partitioning, and skipping and adapts these phases to the streaming mechanisms of PostgreSQL. In any other respect, the module relies on the already available PostgreSQL internals.

The most important native PostgreSQL data structure for the execution of the staircase join is a variant of the B-tree index, the *inner-join index*. As the name implies, it was especially designed to serve as inner relation in a join. Assume a join clause $context.pre < doc.pre$, where *context* is the outer and *doc* the inner relation and *doc* has an index on column *pre*. In this case, a preorder rank p of a node in *context* can be used as index search key to trigger an index scan of *doc* which is guaranteed to start directly at the first tuple with $doc.pre > p$. Since we scan *context* in ascending *pre*-order (Figure 3) and due to partitioning as well as pruning, this leads to a progressive forward scan of *doc*. This also blends perfectly with PostgreSQL's page caching behavior (Section 4).

For the staircase join, we assume that such an index exists on (at least) the *pre* column of the document table. This feature is also crucial for the efficient implementation of skipping.

The original staircase join algorithms [2] materialize the join result. However, since PostgreSQL strives to avoid materialization, the algorithms had to be modified such that each operator in the execution plan only requests the next input tuple from its subplan if immediately required for processing.

The clearly distinguished execution steps predefined by pipelining and the three staircase join-specific techniques (pruning, partitioning, and skipping) suggested the use of a *finite state automaton* to implement the staircase join's execution module. Each of the four major axes was assigned its own automaton.

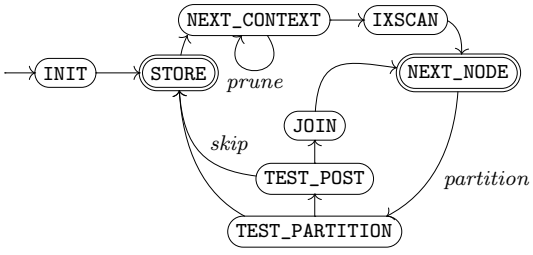


Figure 4: The `descendant` axis state automaton.

The state automaton of the `descendant` axis is outlined in Figure 4. After the first context tuple v_1 has been retrieved from the outer subplan in the `INIT` state, it is stored as lower boundary of the first partition. To identify the upper boundary of the partition (cf. v_2 in Figure 2), the `NEXT_CONTEXT` state continues to request context nodes from the outer subplan, until the next one with a higher post value than v_1 is found (pruning). As soon as the first partition is set, the join starts to retrieve the document nodes within the partition. To do so, a scan of the document table index is initiated. It makes sure that all returned document nodes n have a higher pre value than context node v_1 (`IXSCAN` and `NEXT_NODE`). The `TEST_PARTITION` state verifies that the pre value of n does not exceed the upper partition boundary ($v_2.pre$). If the post clause is also satisfied for v_1 and n, the `JOIN` state can build and return the next result node. If the `TEST_PARTITION` state encounters the first document node outside the current partition, the executor switches to the next partition (`STORE`).

The real benefit of the document table index becomes apparent in connection with skipping. In case of the `descendant` axis, this technique was incorporated into the `TEST_POST` state. If the post clause evaluates to false, we have found the first following node of v_1 (cf. node n in Figure 2 (d)) and may skip the remaining inner tuples in the current partition. The index directly guides us to the first node of the subsequent partition.

The automaton reaches a final state, if either the outer or the inner subplan runs out of tuples.

4 Performance Benefits

To assess the benefits of tree awareness, tests were executed on a 2.2 GHz Dual-Pentium 4 machine with 2 GB RAM. Experiments were run on both, an original and a tree-aware instance of PostgreSQL 7.3.3. The tests examine the buffer-related behavior and the execution times of the example XPath expression Q_2 = `//descendant/ancestor` in dependence on the size of the input XML document (XMark instances of size 110 KB up to 1.1 GB). More experiments were conducted in [3].

The original database chooses two index nested-loop joins to answer Q_2 and evaluates all region query clauses in the index. The execution plan chosen by the tree-aware database is similar to the plan of Figure 3. It evaluates the pre and post clause during the staircase join, the index is exclusively responsible for skipping.

4.1 Execution Times

Figure 5 compares the execution times obtained in both database instances. It shows that the staircase join leads to a performance boost of up to several orders of magnitude. While the execution times of the

original DBMS grow quadratically for this two-step XPath query, those of the enhanced DBMS grow linearly with the document size as expected.

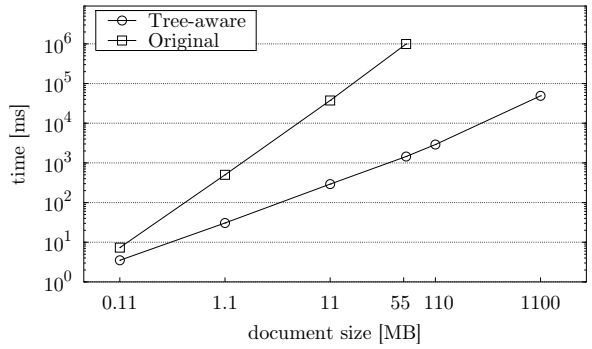


Figure 5: Execution times of Q_2 in the tree-aware and the original PostgreSQL instance.

4.2 Buffer and Cache Behavior

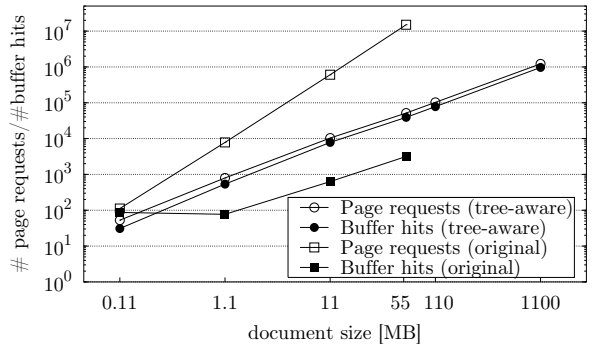


Figure 6: Total number of requested index pages of Q_2 in the tree-aware and the original DBMS ○/□ and buffer hits ●/■ occurred.

Figure 6 shows the buffer statistics of the document index in both databases. The growth in index page requests almost exactly reflects the tendencies observed in Figure 5. We find a quadratic growth in the original and a linear growth in the tree-aware DBMS. This is due to the fact that the staircase join requires exactly one scan of the document table, while the nested-loop join requires $|context|$ scans.

The high number of buffer hits in the tree-aware DBMS is caused by partitioning and the interconnected manner in which the location steps of Q_2 are executed. When the ancestor automaton switches to the next partition, the tuples that make up its boundaries are already in the buffer, because their pages were loaded immediately beforehand by the `descendant` automaton. Thus, when the work on the nodes within the subsequent partition begins, it is very likely that these nodes reside on a disk page already in the buffer.

5 Demonstration Setup

The demonstration features a side-by-side comparison of both, an original and a tree-aware instance of PostgreSQL 7.3.3. Both database systems act as back-ends to a common XPath front-end. This front-end allows for a far more complete set of XPath features than has been outlined here (in particular the supported XPath dialect includes all 13 XPath axes as well as node and name tests).

The front-end compiles an XPath expression into an equivalent SQL query that operates on the *doc* table. This SQL text is then presented to the user as well as shipped to both backends for execution.

Since the efficient management of XML documents of *very large* size is one of the core contributions of database technology in XML processing, both databases are supplied with XMark instances whose size ranges between 110 KB and 1.1 GB (or 5,000 to 50 million nodes).

The demonstration makes use of diagnostic features of PostgreSQL to make the preparation as well as the progess of query execution visible for the user. Hooks are installed in both back-ends to generate a graphical presentation of the chosen query plans (much like in Figure 3). Due to its enhanced query planner, the tree-aware instance relies on 🪜 operators to evaluate XPath location steps, while the original instance will fall back to *sort* and index nested-loop join.

During execution, both backends record timings, page request and cache statistics to provide a detailed graphical post-query feedback (cf. Figures 5 and 6).

Finally, an XML serialization routine hooked into PostgreSQL displays the nodes/subtrees selected by the input XPath expression.

To provide a further point of reference and to exemplify the promising potential of database-supported XML processing, the demonstration additionally evaluates the input XPath expression via a "conventional" main-memory based XPath processor. In anticipation of the live demonstration, the latter class of processors are no match for an RDBMS that has received an injection of tree awareness.

References

[1] Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the 21st ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, pages 109–120. ACM Press, Madison, Wisconsin, USA, June 2002.

[2] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its Axis Steps. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, pages 524–535. Berlin, Germany, September 2003.

[3] Sabine Mayer. *Enhancing the Tree Awareness of a Relational DBMS: Adding Staircase Join to PostgreSQL.* Master's thesis, Universität Konstanz, February 2004.

[4] PostgreSQL 7.3.3. `http://www.postgresql.com/`.