Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps

Torsten Grust° Maurice van Keulen• Jens Teubner°

°University of Konstanz
Department of Computer and Information Science
P.O. Box D 188, 78457 Konstanz, Germany
{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

Abstract

Relational query processors derive much of their effectiveness from the awareness of specific table properties like sort order, size, or absence of duplicate tuples. This text applies (and adapts) this successful principle to database-supported XML and XPath processing: the relational system is made *tree aware*, *i.e.*, tree properties like subtree size, intersection of paths, inclusion or disjointness of subtrees are made explicit. We propose a local change to the database kernel, the *staircase join*, which encapsulates the necessary tree knowledge needed to improve XPath performance. *Staircase join* operates on an XML encoding which makes this knowledge available at the cost of simple integer operations (*e.g.*, $+, \leqslant$). We finally report on quite promising experiments with a *staircase join* enhanced main-memory database kernel.

1 Introduction

Relational database management systems (RDBMSs) have repeatedly shown how versatile the relational data model can be. RDBMSs are successfully used to host types of data which have formerly not been anticipated to live inside relational databases, *e.g.*, non-first normal form (NF^2, nested) tables, complex objects, and spatio-temporal data.

This paper contributes a building block, the *staircase join*, that can help relational technology to also embrace the *tree* data type. We ultimately strive for efficient database support for XML data storage and XPath path queries [2]. The work's key observation is that the RDBMS's efficiency can be significantly improved with an *increasing level of awareness* of the specific properties of the tree data type.

A now quite substantial body of research work has proposed a number of *tree mappings* to encode tree-shaped data, especially XML documents, using relations [5, 8, 11, 16, 17]. These encodings are designed such that tree-specifics, like the *ancestor/descendant* relationship of two tree nodes, can be recovered by a relational query. This level of tree awareness is sufficient to devise efficient relational implementations of the XPath `ancestor`, `parent`, `child`, and `descendant` axes.

In recent former work, we presented an encoding of XML data, the *XPath accelerator* [8], which added real XPath awareness in the sense that all 13 XPath axes could be supported efficiently. In a nutshell, the XPath accelerator uses the *preorder* and *postorder ranks* of document nodes to map nodes onto a two-dimensional plane. The evaluation of XPath axis steps then boils down to process region queries in this *pre*/*post* plane.

This work is orthogonal in that we build upon the simple XPath accelerator structure exactly as it was described in [8], and shift focus to exploit additional properties of the *pre*/*post* plane which help to significantly speed up XPath query evaluation. These formerly unexplored properties exclusively derive from the fact that we encode tree-shaped data. Thus, we increase the level of tree awareness once more.

Pure SQL queries are capable of exploiting some of these properties but not all. The gist of this paper thus is the *staircase join* proposal, a join operator carefully tuned to exploit and encapsulate all "tree knowledge" present in the *pre*/*post* plane. With the staircase join

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

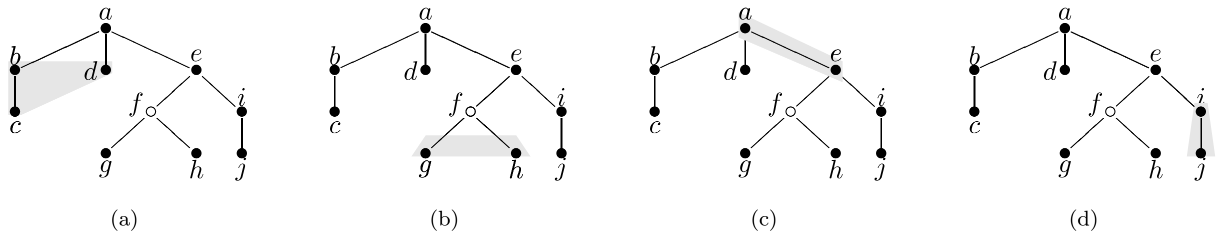


Figure 1: XPath axes induce document regions: shaded nodes are reachable from context node f via a step along the (a) `preceding`, (b) `descendant`, (c) `ancestor`, (d) `following` axes. Leaf nodes denote either empty XML elements, attributes, text, comment, or processing instruction nodes; inner nodes represent non-empty elements.

added to the RDBMS's kernel, SQL again suffices to make full use of the tree awareness now present.

The paper proceeds as follows. Section 2 briefly reviews the core details of the XPath accelerator idea and the limitations an RDBMS faces if its SQL processor is not taught about advanced tree properties. Section 3 introduces the staircase join and discusses the tree-aware optimizations built into the join operator. The *pre*/*post* plane makes it particularly easy to extract certain tree properties. This leads to algorithms which use only a few CPU cycles per XML document node. We discuss the implications for the implementation of staircase join, especially in the context of the relational main-memory DBMS *Monet* in Section 4. The section closes with a performance assessment of the ideas developed so far. Section 5 reviews related research before we conclude in Section 6.

2 The XPath Accelerator

The *XPath accelerator* [8] is a *relational* XML document encoding. Here, *relational* is meant in the sense of [10]: the encoded document (1) is represented as a relational table, (2) can and should be indexed using index structures native to the RDBMS (preferably B-trees, see below), and (3) may be queried using a relational language, namely SQL.

The encoding has been designed with a close eye on the XPath axes semantics: for each node of a given XML document, the four axes `preceding`, `descendant`, `ancestor`, and `following` partition the document into four disjoint *regions*. Figure 1 depicts the document regions for a 10-node XML instance as seen from *context node* f. The XPath expression f`/preceding::node()`—abbreviated as f`/preceding` from now on—yields the node sequence (b, c, d).

Note that the context node plus the nodes in the four regions cover *all* document nodes. The XPath accelerator makes use of this observation: the nodes of a document are encoded such that the region notion is maintained. The evaluation of an XPath step along those four axes then amounts to retrieving the nodes contained in the region associated with that axis.[1] All further XPath axes determine easily characterizable super- or subsets of these regions (*e.g.*, `ancestor-or-self`) or are supported by standard RDBMS join algorithms (*e.g.*, `child`, `parent`) [8]. The focus of this paper will, therefore, be on the four partitioning axes.

The actual encoding maps each node v to its *preorder* and *postorder traversal rank* in the document tree:

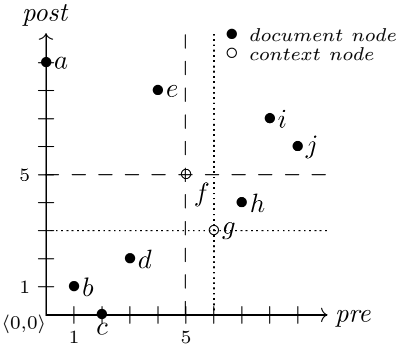
$$v \mapsto \langle pre(v), post(v) \rangle \ .$$

Figure 2 shows the two-dimensional *pre*/*post* plane that results from encoding the sample XML instance of Figure 1. Like the original document tree, the *pre*/*post* plane is partitioned into four, now rectangular, regions which characterize the XPath `preceding`, `descendant`, `ancestor`, and `following` axes, *e.g.*, the nodes f`/preceding` $= (b, c, d)$ are located in the lower left region with respect to context node f.

Note that this characterization of document regions applies to *all* nodes in the plane. For example, the upper left region with respect to g hosts the nodes g`/ancestor` $= (a, e, f)$. This means that we can pick any node v and use its location in the plane to evaluate an XPath step, *i.e.*, make v the context node. This turns out to be an important feature when it comes to the implementation of, *e.g.*, XQuery [3], where expressions compute arbitrary context nodes and then traverse from there. Exactly this usage scenario led to the development of the present ideas: the XPath accelerator serves as the back-end of the *Pathfinder* XQuery compiler runtime currently under development at the University of Konstanz.

We refer to [8] for a more detailed explanation of the XPath accelerator idea.

[1] In what follows we assume that a database stores a single (large) document. Our discussion readily carries over to multi-document databases (*e.g.*, by introduction of *document identifiers* or a new *virtual root node* under which several documents may be gathered.)



	pre	*post*
a	0	9
b	1	1
c	2	0
d	3	2
e	4	8
f	5	5
g	6	3
h	7	4
i	8	7
j	9	6

Figure 2: *Pre/post* plane and the correspoding node encoding table doc for the XML document of Figure 1. Dashed and dotted lines indicate the document regions as seen from context nodes f (– –) and g (·········), respectively.

```
1 SELECT DISTINCT v2.pre
2   FROM doc v1 ,doc v2
3  WHERE v1.pre > pre(c)
4    AND v1.pre < v2.pre
5    AND v1.post > post(c)
6    AND v1.post > v2.post
8  ORDER BY v2.pre
```

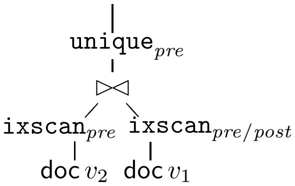


Figure 3: Query and associated plan.

2.1 SQL-based XPath Evaluation

The *pre/post* plane encoding enables an RDBMS to translate XPath path expressions to pure SQL queries.

The evaluation of an XPath path expression[2] like s_1/s_2/ $\cdots$ /s_n leads to a series of n region queries in the *pre/post* plane where the node sequence output by axis step s_i is the context node sequence for the subsequent step s_{i+1}. (The context node sequence for step s_1 is a singleton sequence (v) with v being an arbitrary node.) Note that the XPath semantics require the resulting node sequence to be *duplicate free* as well as being sorted in *document order* [2].

In the *pre/post* plane of Figure 2, with initial context node sequence (c) and XPath path expression `following::node()/descendant::node()`, we get

$$(c)/\texttt{following/descendant} = (f, g, h, i, j) \enspace .$$

Let doc denote the two-column table loaded with the *pre/post* node encodings (*cf.* Figure 2), then we can systematically translate a path expression into an equivalent SQL query [8]. For the example above, we get the query in Figure 3.

An analysis of the actual query plan chosen by the optimizer—IBM DB2 V7.1 in this case—shows that a relational database system can cope quite well with this type of query (Figure 3): the RDBMS maintains a B-tree using concatenated (*pre*, *post*) keys. The index is used to scan the outer (left) doc table in *pre*-sorted order. The actual region query evaluation happens in the inner join input: the predicates in lines 3 and 4 of the SQL query act as index scan range delimiters while the predicates in lines 5 and 6 are sufficiently simple to be evaluated during the B-tree index scan as well. The join is actually a left semijoin producing its output in *pre*-sorted order (which matches the request in line 8 for a result sorted in *document order*).

Actually, the query optimizer could further delimit the index range scan in the inner join input. This opportunity derives from the fact that in any tree and for any node v we can anticipate the size of the subtree below v:

$$|\,(v)/\texttt{descendant}\,| = post(v) - pre(v) + \underbrace{level(v)}_{\leqslant h} \enspace , \tag{1}$$

where $level(v)$ denotes the length of the path from the root to v which is obviously bound by h, the *height* of the overall document tree.[3] With this "tree knowledge" available to the RDBMS, the optimizer could delimit the descendant range scan [8] and insert an additional predicate into the initial SQL query:

```
7  AND v2.pre ≤ v1.post + h AND v2.post ≥ v1.pre + h .
```

The inner index range scan for the descendant step is now delimited by the actual size of the context nodes' subtrees and independent of the document size. In [8] we observed a query speed-up of up to three orders of magnitude. We will see in the sequel that awareness of facts like Equation (1) can lead to more significant improvements in XPath performance.

Notice that the unique operator in the plan of Figure 3 is indeed required since, in general, the join will generate duplicate nodes (see Section 3.1). The generation of duplicates and thus the rather costly unique operator, however, could be avoided altogether if the join operator would be informed about the fact that the doc table encodes a tree structure. For similar reasons, we furthermore could improve on the ixscans: since the node distribution in *pre/post* plane is not arbitrary, we can actually *skip* significant portions and guide the scans to touch only those nodes which constitute the actual result (modulo a small misestimation).

[2] The XPath accelerator supports further XPath features like predicates, node tests, *etc.*, as well [8].

[3] The system computes h at document loading time. For typical real-world XML instances we found $h \approx 10$.

This and further knowledge *is* present in *pre*/*post* encodings although not accessible to the query optimizer unless it can be made explicit at the SQL level (as it is the case with the additional range predicate in line 7 above).

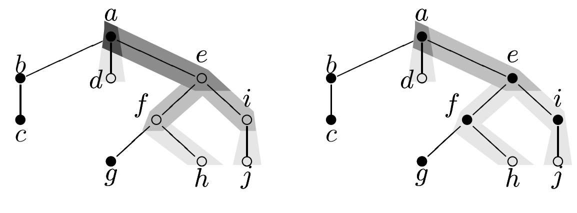
3 The Staircase Join

Making the query optimizer of an RDBMS more "tree aware" would allow it to improve its query plans concerning XPath evaluation. However, incorporating knowledge of the *pre*/*post* plane should, ideally, not clutter the entire query optimizer with XML-specific adaptations. As explained in the introduction, we propose a special join operator, the *staircase join*, that exploits and encapsulates all "tree knowledge" present in the *pre*/*post* plane. It behaves to the query optimizer in many ways as an ordinary join, for example, by admitting selection pushdown. In this section, we will describe the staircase join and the tree-aware optimizations it encapsulates.

Before we proceed, a note on attributes. Except for the `attribute` axis itself, no axis produces attribute nodes. We use a special encoding for attribute nodes, which allow them to be filtered out if needed. We disregard attributes in our explanations, however, because it would clutter them unnecessarily. Given numbers and experimental results obviously include attribute handling. Whenever the effects of attribute handling are observable, we indicate this via footnotes.

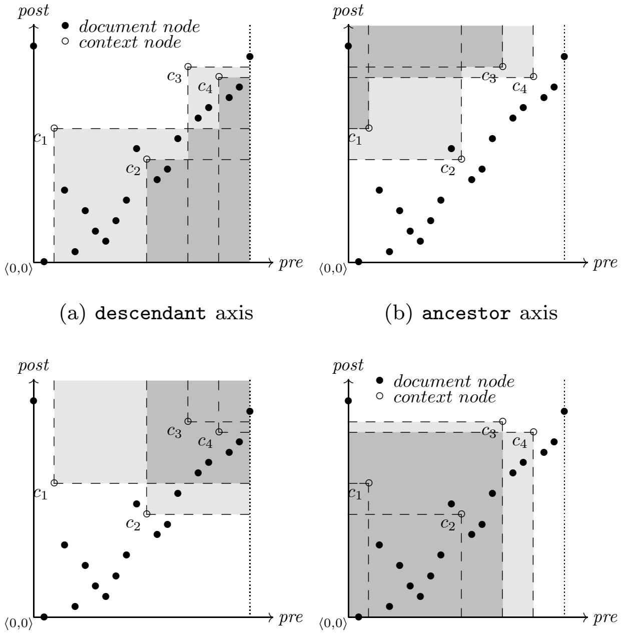
3.1 Pruning

The evaluation of an axis step for a certain context node boils down to selecting all document nodes in the corresponding region. In XPath, however, an axis step is generally evaluated on an *entire sequence* of context nodes [2]. This leads to duplication of work if the *pre*/*post* plane regions associated with the step are independently evaluated for each context node. Figure 4 (a) depicts the situation if we are about to evaluate an `ancestor-or-self` step for context sequence (d, e, f, h, i, j). The darker the path's shade, the more often are its nodes produced in the resulting node sequence—which ultimately leads to the need for duplicate removal operator `unique` in the query plan of Figure 3 to meet the XPath semantics. Obviously, we could remove nodes e, f, i—which are located along a path from some other context node up to the root—from the context node sequence without any effect on the final result (a, d, e, f, h, i, j) (Figure 4 (b)). Such opportunities for the simplification of the context node sequence arise for all axes.



(a) (b)

Figure 4: (a) Intersection and inclusion of the `ancestor-or-self` paths of a context node sequence. (b) The pruned context node sequence covers the same `ancestor-or-self` region and produces less duplicates (3 rather than 11).



(a) `descendant` axis (b) `ancestor` axis

(c) `following` axis (d) `preceding` axis

Figure 5: Overlapping regions (context nodes c_i).

Figure 5 depicts the situation in the *pre*/*post* plane as this is the RDBMS's view of the problem (these planes show the encoding of a slightly larger XML document instance). For each axis, the context nodes establish a different boundary enclosing a different area. Result nodes can be found in the shaded areas. In general, regions determined by context nodes can *include* one another or *partially overlap* (dark areas). Nodes in these areas generate duplicates.

The removal of nodes e, f, i earlier is a case of *inclusion*. Inclusion can be dealt with by removing the covered nodes from the context: for example, c_2, c_4 for (a) `descendant` and c_3, c_4 for (c) `following` axis.

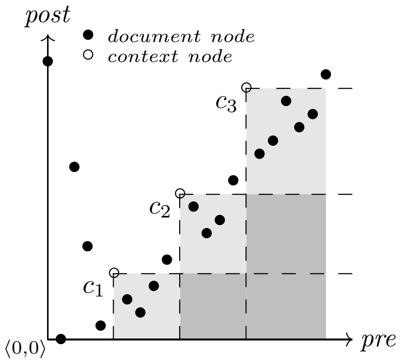


Figure 6: Pruning produces a proper staircase.

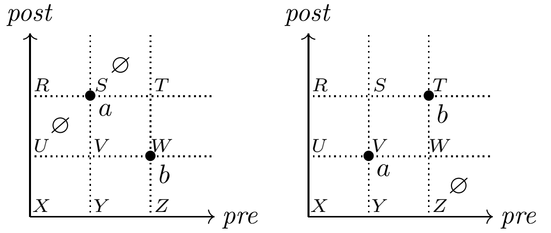
The process of identifying the context nodes at the cover's boundary is referred to as *pruning* and is easily implemented for a *pre/post* encoded context node sequence. Algorithm 1 gives the pruning procedure for the `descendant` axis (`ancestor` pruning functions analogously).

```
prunecontext_desc (context : TABLE (pre,post)) ≡
  BEGIN
    result ← NEW TABLE (pre, post); prev ← 0;
    FOREACH c IN context DO
      IF c.post > prev THEN
        APPEND c TO result;
        prev ← c.post;
    RETURN result;
  END
```

Algorithm 1: Context pruning for `descendant` axis, table `context` is assumed to be *pre*-sorted.

After pruning for the `descendant` or `ancestor` axis, all remaining context nodes relate to each other on the `preceding/following` axis as illustrated for `descendant` in Figure 6. The context establishes a boundary in the *pre/post* plane that resembles a staircase.

Observe in Figure 6 that the three darker subregions do not contain any nodes. This is no coincidence. Any two nodes a, b partition the *pre/post* plane into nine regions R through Z (see Figure 7). There are two cases to be distinguished regarding how both nodes relate to each other: (a) on `ancestor/descendant` axis or (b) on `preceding/following` axis. In (a), regions S, U are necessarily empty because an ancestor of b cannot precede (region U) or follow a (region S) if b is a descendant of a. Similarly, region Z in (b) is empty, because a, b cannot have common descendants if b follows a. The empty regions in Figure 6 correspond to such Z regions. An RDBMS that is tree-aware enough to know about pruning and empty regions, could use this knowledge to further delimit index range scans and thus exclude the darker regions.



(a) Nodes a and b relate to each other on the `ancestor/descendant` axis.

(b) Nodes a and b relate to each other on the `preceding/following` axis.

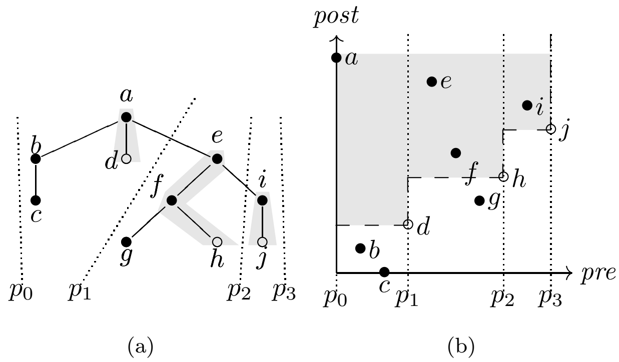
Figure 7: Empty regions in the *pre/post* plane.

A similar empty region analysis can be done for all XPath axes. The consequences for the `preceding` and `following` axes are more profound. After pruning for, *e.g.*, the `following` axis, the remaining context nodes relate to each other on the `ancestor/descendant` axis. In Figure 7 (a), we see that for any two remaining context nodes a and b, (a, b)`/following` $= S \cup T \cup W$. Since region S is empty, (a, b)`/following` $= T \cup W =$ (b)`/following`. Consequently, we can prune a from the context (a, b) without affecting the result. If this reasoning is followed through, it turns out that all context nodes can be pruned except the one with the maximum preorder rank in case of `preceding` and the minimum postorder rank in case of `following`. For these two axes the context is reduced to a singleton sequence such that the staircase join degenerates to a single region query. We will therefore focus on the `ancestor` and `descendant` axes in the following.

3.2 Basic Staircase Join Algorithm

While pruning leads to a significant reduction of duplicate work, Figure 4 (b) exemplifies that duplicates still remain due to intersecting `ancestor-or-self` paths originating in different context nodes. A much better approach results if we *separate* the paths in the document tree and evaluate the axis step for each context node in its own partition (Figure 8 (a)).

Such a separation of the document tree is easily derived from the staircase induced by the context node sequence in the *pre/post* plane (Figure 8 (b)): each of the partitions $[p_0, p_1)$, $[p_1, p_2)$, and $[p_2, p_3)$ define a region of the plane containing all nodes needed to compute the axis step result for context nodes d, h, and j, respectively. Note that pruning reduces the number of these partitions. (Although a review of the details is outside the scope of this text, it should be obvious



(a)

(b)

Figure 8: The partitions $[p_0, p_1)$, $[p_1, p_2)$, $[p_2, p_3)$ of the `ancestor` staircase separate the `ancestor-or-self` paths in the document tree.

that the partitioned *pre*/*post* plane naturally leads to a parallel XPath execution strategy.)

The basic approach to evaluating a staircase join between a document and a context node sequence thus is to sequentially scan the *pre*/*post* plane once from left to right selecting those nodes in the current partition that lie within the boundary established by the context node sequence (see Algorithm 2). Since the XPath accelerator maintains the nodes of the *pre*/*post* plane in the *pre*-sorted table doc, staircase join effectively visits the tree in document order. The nodes of the final result are, consequently, encountered and written in document order, too.

The basic algorithm is perhaps most closely described as a merge join with a dynamic range predicate. It is important to observe that 'doc[i].*post*' in the algorithm is not really a distinct lookup for the record with preorder rank i, since the record is encountered during a sequential scan. The notation 'doc[i]', hence, just means *the record at hand*. Note, furthermore, that the algorithm only works correctly on proper staircases, *i.e.*, with an already pruned context. Although we presented pruning as a separate pre-processing stage, staircase join is easily adapted to do pruning *on-the-fly*, thus saving a separate scan over the context table.

This basic algorithm has several important characteristics:

(1) it scans the doc and context tables sequentially,
(2) it scans both tables only once for an entire context sequence,
(3) it never delivers duplicate nodes, and
(4) result nodes are produced in document order, so no post-processing is needed to comply with XPath semantics.

```
staircasejoin_desc (doc : TABLE (pre,post),
                    context : TABLE (pre,post)) ≡
  BEGIN
    result ← NEW TABLE (pre, post);
    FOREACH SUCCESSIVE PAIR ($c_1, c_2$) IN context DO
      scanpartition ($c_1$.pre + 1, $c_2$.pre − 1, $c_1$.post,<);
    c ← LAST NODE IN context;
    n ← LAST NODE IN doc;
    scanpartition (c.pre + 1, n.pre, c.post,<);
    RETURN result;
  END
staircasejoin_anc (doc : TABLE (pre,post),
                   context : TABLE (pre,post)) ≡
  BEGIN
    result ← NEW TABLE (pre, post);
    c ← FIRST NODE IN context;
    n ← FIRST NODE IN doc;
    scanpartition (n.pre, c.pre − 1, c.post,>);
    FOREACH SUCCESSIVE PAIR ($c_1, c_2$) IN context DO
      scanpartition ($c_1$.pre + 1, $c_2$.pre − 1, $c_2$.post,>);
    RETURN result;
  END
scanpartition ($pre_1$,$pre_2$,post,$\theta$) ≡
  BEGIN
    FOR i FROM $pre_1$ TO $pre_2$ DO
      IF doc[i].post $\theta$ post THEN
        APPEND doc[i] TO result;
  END
```

Algorithm 2: Staircase join algorithms (`descendant` and `ancestor` axes).

3.3 More Tree-Aware Optimization: Skipping

The empty region analysis explained in Section 3.1 offers another kind of optimization, which we refer to as *skipping*. Figure 9 illustrates this for the XPath axis step (c_1, c_2)/`descendant`. The staircase join is evaluated by scanning the *pre*/*post* plane from left to right starting from context node c_1. During the scan of c_1's partition, v is the first node encountered outside the `descendant` boundary and thus not part of the result.

Note that no node beyond v in the current partition contributes to result (the light grey area is empty). This is, again, a consequence of the fact that we scan the encoding of a tree data structure: node v is following c_1 in document order so that both cannot have common descendants, *i.e.*, the empty region in Figure 9 is a region of type Z in Figure 7 (b).

Staircase join uses this observation to terminate the current `scanpartition` early which effectively means that the portion of the scan between *pre*(v) and the successive context node *pre*(c_2) is *skipped*.

The change to the basic staircase join algorithm is minimal as only procedure `scanpartition` is af-

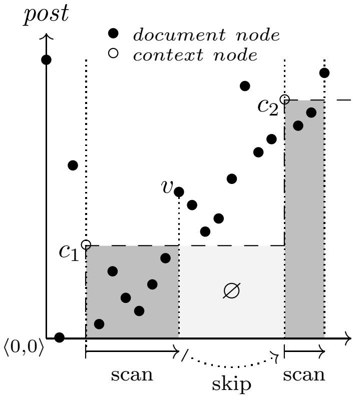


Figure 9: Skipping technique for `descendant` axis.

fected. Procedure `staircasejoin_desc` is merely modified to invoke the new replacement procedure `scanpartition_desc` shown as Algorithm 3.

```
scanpartition_desc ($pre_1$,$pre_2$,$post$) ≡
  BEGIN
     FOR $i$ FROM $pre_1$ TO $pre_2$ DO
(⋆)     IF doc[$i$].$post < post$ THEN
           APPEND doc[$i$] TO result;
        ELSE
           BREAK;    /* skip */
  END
```

Algorithm 3: Skipping for the `descendant` axis.

The effectiveness of skipping is high. For each node in the context, we either (1) hit a node to be copied into the result, or (2) encounter a node of type v which leads to a skip. To produce the result, we thus never touch more than |result| + |context| nodes in the *pre/post* plane while the basic Algorithm 2 would scan along the entire plane starting from the context node with minimum preorder rank.

A similar, although slightly less effective skipping technique can be applied to the `ancestor` axis: if, inside the partition of context node c, we encounter a node v outside the `ancestor` boundary, we know that v as well as all descendants of v are in the *preceding* axis of c and thus can be skipped. In such a case, Equation (1) provides us with a good estimate—which is maximally off by the document height h—of how many nodes we may skip during the sequential scan, namely $post(v) - pre(v)$.

4 Main-Memory RDBMS Implementation Considerations

To assess the viability and the impact of the staircase join idea, we enhanced the kernel of the *Monet* RDBMS [4] to incorporate the new join operator. The main-memory RDBMS Monet has been chosen as the preferred implementation platform for the aforementioned *Pathfinder* project. Additionally, Monet's base type and operator extensibility make the system a suitable experimentation platform. Adding staircase join to a main-memory RDBMS kernel allowed us to study CPU-related and cache-related adaptations to the original join algorithms. It turns out that staircase join can be optimized well for in-memory operation. We close this section with a number of experiments to manifest the efficiency of staircase join.

4.1 Experimentation Platform

We first describe our experimentation platform, Monet, in order to have some concrete material for illustrative purposes. We then show that CPU-related and cache-related adaptations to the staircase join algorithm are possible.

First off, Monet's bulk data type, the binary table (*BAT*), is a good match for the two-column table doc holding the *pre/post* document encoding. BATs provide several useful features, like the special column type `void`: *virtual oid*. A column of this type represents a contiguous sequence of integers $o, o{+}1, o{+}2, \ldots$ for which only the offset o needs to be stored. This not only saves storage space—a document occupies only about 1.5× its size in Monet using our storage structure—it also allows many operations to be executed using positional lookups. For more details about the Monet RDBMS, we refer to [4].

In our experiments, we used a Dual–Pentium 4 (Xeon) machine running on $2.2\,GHz$, $2\,GB$ main-memory, a two-level cache (levels L_1/L_2) of size $8\,kB/512\,kB$, L_1/L_2 cache line size $32\,byte/128\,byte$, L_1/L_2 miss latency $28\,cy/387\,cy = 12.7\,ns/176\,ns$ (measured with *Calibrator* [12]). Without loss of generality, we will use the characteristics of this machine to illustrate machine-dependent calculations.

4.2 CPU-related Adaptations

The staircase join algorithm basically includes two loops which scan the context and doc BATs, respectively. The context sequence ordinarily contains far less elements than the document, so we concentrate on the inner loop `scanpartition_desc` (Algorithm 3). It sequentially scans a given partition of the doc BAT. Each iteration contains a comparison and a write to the result BAT (except for the last iteration).

The preorder ranks in table doc form a contiguous sequence. We use Monet's `void` column type and thus only store (and scan) the postorder ranks of $4\,byte$ each. An L_2 cache line, hence, contains

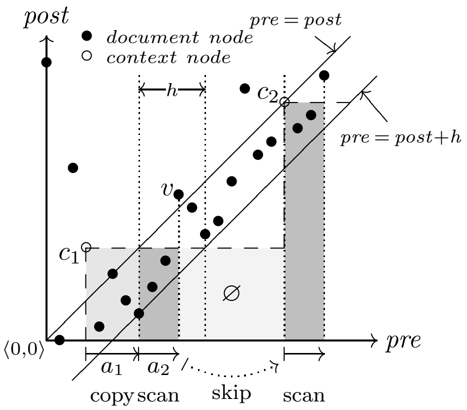


Figure 10: Estimation-based skipping (h = height of document).

$^{128}/_{4} = 32$ nodes. On this machine, CPU work for one iteration in `scanpartition_desc` is about $17\,cy$.[4] For one cache line, this is $17\,cy \times 32 = 544\,cy$ which exceeds the L_2 miss latency of $387\,cy$. Therefore, Algorithm 3 is CPU-bound so we concentrate on reducing CPU work first.

A major part of the CPU work concerns the postorder rank comparison in the line labeled ($\star$) in Algorithm 3. For a large part, we can take it out of the main loop as follows. According to Equation (1) on page 3, we can calculate a lower and upper bound for the number of descendants of a node v ($0 \leqslant level(v) \leqslant h$). These bounds establish two diagonals in the *pre*/*post* plane (see Figure 10).

Take context node c_1. Because a preorder traversal of a tree encounters the descendants of c_1 directly after c_1 itself has been visited, it is guaranteed that the first $post(c_1) - pre(c_1)$ nodes following c_1 in the *pre*/*post* plane are its descendants. Consequently, we can simply *copy* these nodes to result without checking their postorder ranks (*copy phase*; interval a_1 in the figure). The upper bound tells us that there are at most h additional descendants, which we can obtain using the original loop (*scan phase*; interval a_2 in the figure). As before, we stop scanning at the first following node found and *skip* to the next context node. We call this technique *estimation-based skipping*, because we estimate the number of descendants *before* we enter the tight copy loop. See Algorithm 4.

The error of our estimation is maximally h.[5] Therefore, we have restricted postorder rank comparison to at most $h \times |\mathsf{context}|$ nodes. The remaining nodes, at least $|\mathsf{result}| - (h \times |\mathsf{context}|)$, are simply copied. A single node copy iteration takes about 5 cycles. Processing one L_2 cache line now takes $5\,cy \times 32 = 160\,cy$ which clearly undercuts L_2 miss latency. The copy phase is therefore cache-bound rather than CPU-bound. Since h is small (in the order of tens of nodes, insignificant in multi-million node documents), the copy phase represents the bulk of the work.

```
scanpartition_desc ($pre_1$,$pre_2$,$post$) ≡
  BEGIN
    $estimate \leftarrow$ min($pre_2$, $post$);
    /* copy phase */
    FOR $i$ FROM $pre_1$ TO $estimate$ DO
      APPEND doc[$i$] TO result;
    /* scan phase */
    FOR $i$ FROM $estimate + 1$ TO $pre_2$ DO
      IF doc[$i$].$post < post$ THEN
        APPEND doc[$i$] TO result;
      ELSE
        BREAK;   /* skip */
  END
```

Algorithm 4: Estimation-based skipping (`descendant` axis).

The low number of CPU cycles is also due to the *branch-prediction friendliness* of staircase join. The loops of both phases have a fixed end condition and the IF conditional always chooses the same (THEN) branch except for the last iteration. The branching behavior of the tight inner loops of staircase join is perfectly predictable such that the significant penalties for instruction retirement are avoided [14]. We now shift our attention to the CPU cache, as it turned out to be the bottleneck in the copy phase.

4.3 Cache-related Adaptations

The sequential scan of the *pre*/*post plane* is main-memory friendly since CPU cache lines are fully used. The sequential memory bandwidth of a machine with two cache levels can be calculated as follows [4]:

$$\begin{aligned}\frac{LS_{L_2}}{L_{L_2} + \dfrac{LS_{L_2}}{LS_{L_1}} \times L_{L_1}} &= \frac{128\,byte}{176\,ns + \dfrac{128\,byte}{32\,byte} \times 12.7\,ns} \\ &= 551\,{}^{MB}/_{s}\end{aligned}$$

(with LS_C = cache line size of cache C; L_C = cache miss latency for cache C).

Pentium 4 like other modern processors, however, supports *hardware prefetching* which partially hides the memory and cache latency effects: after a startup

[4] This number has been computed from the actual assembler instruction latencies of Pentium 4 [6].

[5] We have devised an alternative pre/postorder encoding that allows exact calculation of $|(v)$`/descendant`$|$ for any node v. As this paper builds upon [8], we used the original encoding described there.

penalty—when the CPU has recognized the purely sequential access pattern of staircase join—it will read two L_2 cache lines ($= 256\,\textit{byte}$) ahead.

The copy phase uses two data streams (Pentium 4 supports up to 8 independent streams): one load stream (doc) and one store stream (result). The simple experiment of evaluating $(\textit{root})$`/descendant` can show that hardware prefetching indeed improves performance of the staircase join. This particular experiment consists almost entirely of a copy phase (for query characteristics, see Table 1). It showed a bandwidth of

$$\begin{aligned}
&\frac{\textit{bytes read} + \textit{bytes written}}{\textit{execution time}} \\
&= \frac{(|\mathsf{doc}| + \mathsf{context}\ \textit{nodes scanned} + \textit{result size}) \times 4\,\textit{byte}}{\textit{execution time}} \\
&= \frac{(50,844,982 + 1 + 47,015,212) \times 4\,\textit{byte}}{519\,\textit{ms}} \\
&= 719.0\,{}^{\textit{MB}}\!/_{\textit{s}}\ .
\end{aligned}$$

Intel suggests [6] that further bandwidth improvement can be obtained by employing *software prefetching*. Given the overall $L_1 + L_2$ miss latency, $28\,\textit{cy} + 387\,\textit{cy} = 415\,\textit{cy}$, it is suggested to additionally put an explicit prefetch-instruction (`prefetchnta`) in the algorithm to prefetch 3 cache lines ahead. In combination with extra loop unrolling (reducing loop overhead) and employing *Duff's device* [7], the bandwidth was boosted to $805\,{}^{\textit{MB}}\!/_{\textit{s}}$.

Although the numbers given here are specific for our platform, we believe a staircase join implementation in another RDBMS may encounter similar conditions and may admit similar optimizations.

4.4 Experiments

The staircase join encapsulates "tree knowledge" by applying the pruning and skipping techniques described in Sections 3 and 4. In the remainder of this section, we will assess the performance gain achieved by employing these techniques.

The system used for the experiments is the one described in Section 4.1. To ensure the test runs to be reproducible, we used an easily accessible source of XML documents, namely the XML generator *XMLgen*, developed for the *XMark benchmark project* [15]. For a fixed DTD, this generator produces instances of controllable size. We have used XML instances of 1 *MB* up to 1 *GB* size (50 000–50 000 000 document nodes). All documents were of height 11. We chose two queries that generate substantial intermediary results (see Table 1; sizes for other documents are proportionally smaller). Both queries use two axis steps: a `descendant` step from the root and a subsequent `descendant` or `ancestor` step. A third, fourth, or further step would behave much like the second axis step which is why we restricted these experiments to two-step paths. Furthermore, we concentrate on `descendant` and `ancestor` as explained earlier. The queries are evaluated as illustrated here for $Q2$:

$$\begin{aligned}
r &= \texttt{root}(\mathsf{doc}) \\
s_1 &= \texttt{nametest}(\texttt{staircasejoin\_desc}(\mathsf{doc}, r), \texttt{"increase"}) \\
s_2 &= \texttt{nametest}(\texttt{staircasejoin\_anc}(\mathsf{doc}, s_1), \texttt{"bidder"})
\end{aligned}$$

$Q1$:	/descendant	::profile	/descendant	::education
	47,015,212[6]	127,984	1,849,360	63,793
$Q2$:	/descendant	::increase	/ancestor	::bidder
	47,015,212	597,777	706,193	597,777

Table 1: Number of nodes in intermediary results (1 *GB* document; 50,844,982 nodes)

Experiment 1: Pruning, Avoiding Duplicates

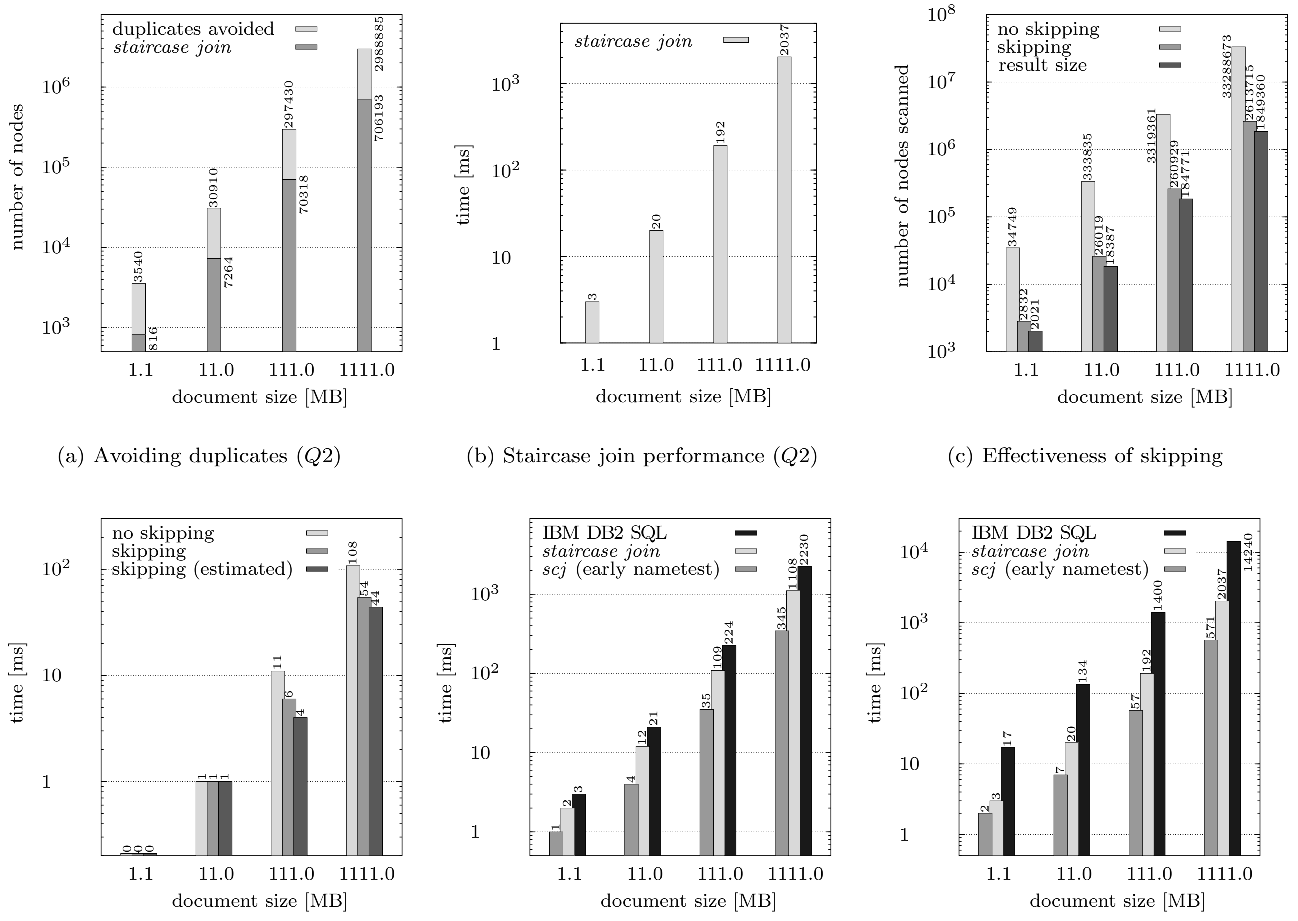
The naive way of evaluating an axis step for a context node sequence would be to evaluate the step for each context node independently and construct the end result from these intermediary results (Section 3.1). One of the advantages of the basic algorithm given in Section 3.2, is that it avoids the generation of any duplicates that the naive approach would produce. Figure 11 (a) shows the number of result nodes that the `ancestor` step in query $Q2$ (excluding the name test) would produce using the naive approach and the staircase join. In this experiment, the staircase join saves generation and subsequent removal of the about 75% duplicates. This number is no coincidence, because the context sequence contains `increase` nodes, which all appear on a path of length 4 up to the root, *i.e.*, for all context nodes c, $\textit{level}(c) = 4$. A large number of pairs of these paths, however, intersect at level 3 leading to a duplicate/node ratio of about ${}^3\!/_4$ (*cf.* Figure 4).

The fact that the staircase join algorithm scans document and context tables sequentially and only once has, besides avoiding duplicates, other advantageous consequences: execution times are linear with document size (see Figure 11 (b)). Moreover, a further effect of the access pattern is that the result is immediately in document order, so a costly sorting phase is avoided.

Experiment 2: Effectiveness of Skipping

Figures 11 (c) and (d) assess the effectiveness of the skipping and estimation-based skipping techniques. The experiment counts accessed nodes and execution times for the staircase join in the second axis step of

[6] This number is smaller than the total number of nodes in the document, since the result of a `descendant` step does not contain any attributes.



(a) Avoiding duplicates ($Q2$)

(b) Staircase join performance ($Q2$)

(c) Effectiveness of skipping

(d) Effectiveness of skipping

(e) Performance comparison ($Q1$)

(f) Performance comparison ($Q2$)

Figure 11: Experimental results (diagrams use a log scale).

query $Q1$. Skipping reduces the number of nodes accessed to at most $|\mathsf{result}| + |\mathsf{context}|$ (see Section 3.3). The logarithmic scale of (c) clearly shows that the amount of nodes accessed for "skipping" that do not appear in the result remains limited.[7] The amount of nodes actually skipped (the difference between "no skipping" and "skipping") is significant: about 92 % of the nodes were skipped. This number obviously depends on the query, but the main point is that skipping makes the number of accessed nodes independent of the document size.

The reduction in accessed nodes has its effect on execution times. Under the same conditions (staircase join in second axis step of query $Q1$), execution time is about cut in half ("no skipping" *vs.* "skipping" for the larger document sizes in (d)). Employing the cache conscious implementation of estimation-based skipping (see Sections 4.2 and 4.3) gives an additional performance gain of about 20% ("skipping (estimated)").

Experiment 3: Comparison with IBM DB2

The focus of this paper is on the staircase join and its benefit for evaluating axis steps. Other aspects of XPath like name tests and predicates, have largely been left out of the discussion. For this reason, an axis step like cs/`ancestor::`n has up to now been treated as `nametest`(`staircasejoin_anc`(`doc`, cs), n), *i.e.*, first fully evaluate the `ancestor` step for context sequence cs, and then do a subsequent name test for tag name n.

In the introduction, however, we already mentioned that the staircase join behaves to the query optimizer in

[7] Because of attribute filtering, the difference between "skipping" and "result size" is slightly larger than $|\mathsf{context}|$. The statement 'we scan at most $|\mathsf{context}|$ nodes too many' holds nevertheless.

many ways as an ordinary join, for example, by admitting selection pushdown. Observe that the result of the staircase join contains a selection of nodes from the `doc` table. The subsequent name test is a selection on tag name n. Pushing the name test through the staircase join, hence, means doing both selections in opposite order. The tree properties used by the staircase join are entirely based on preorder and postorder ranks. Those properties remain valid for a subset of nodes. Therefore, `staircasejoin_anc`(`nametest`(`doc`, n), cs) is a valid equivalent of the example. Note that, consequently, the name test is performed on the entire document, which obviously makes sense for selective name tests only.

Figures 11 (e) and (f) compare execution times for both queries with and without name test pushdown. In both queries, an execution plan with name test pushdown shows to be faster by about a factor 3. Future research on a cost model is intended to let the system intelligently decide for or against name test pushdown or similar rewrites.

The experiment furthermore shows the execution times of an implementation with limited tree awareness (on top of IBM DB2). Considering the performance increases we have seen for avoiding duplicates, pruning, and skipping, we believe a conventional RDBMS can achieve an increase of similar magnitude by employing the staircase join which is in line with the observations in [17]. Note that, although not explicitly shown in the query plan of Figure 3, IBM DB2 also performs an early name test (the B-tree index actually uses concatenated (*pre*, *post*, *tag name*) keys). As a further note, being tree-unaware, an RDBMS query optimizer sometimes makes bad estimations and consequently chooses bad query plans, which happened for our query $Q2$ (this has been observed by others as well [16]): the actual execution times shown are for the SQL query corresponding with the equivalent manual rewrite [13] of $Q2$, `/descendant::bidder[descendant::increase]`.

5 More Related Research

Recall that the core operation of staircase join resembles a self merge join of the *pre*-sorted `doc` table with the join predicate dynamically changed to trace the staircase boundary (*cf.* Section 3.2). In [17], the *multi-predicate merge join* (MPMGJN) was introduced with the specific purpose to efficiently support interval containment predicates. While such predicates allow to express the semantics of the XPath `ancestor` and `descendant` axes, MPMGJN has been designed to exploit the hierarchical containment of intervals and thus lacks further tree awareness: due to pruning and skipping, staircase join touches and tests less nodes than MPMGJN. Nevertheless, we subscribe to the view expressed in [16, 17], that increased awareness of data type properties will be critical in turning RDBMSs into efficient XML processors.

Staircase join is a real self join in the sense that context nodes *and* document nodes are tuples of the `doc` encoding table. A *single* B+-tree—built at document loading time—suffices to index both, arbitrary context sequences and the document. This is unlike the approaches described in [5] and [9], where a special purpose index structure needs to be built over context and document (referred to as *ancestor* and *descendant lists* in [5, 9]) to support `ancestor` or `descendant` step evaluation with skipping. Like staircase join, the algorithms of [5, 9] depend on the lists being sorted in document order. The join operators of [5] are based on a modified B+-tree implementation that uses extra sibling pointers. The authors of [9] propose to build new index structures, the *XR-Trees*, over context and document.

Staircase join derives all pruning and skipping information from the `doc` table itself and it does so using simple integer arithmetic. We do not require the underlying RDBMS's B+-tree implementation to be altered. Furthermore, since a single B+-tree instance indexes the context as well as the *pre*/*post* plane, it is likely that less index pages compete for buffer slots during query evaluation.

The conceptual simplicity of staircase join led to algorithms which exhibit particularly simple control flow. Just like [14], we found this to be critical to achieve high efficiency, especially in the main-memory RDBMS context. Highly predictable branches in its inner loop and a strictly sequential access pattern [1] make staircase join CPU- and cache-friendly.

Finally, note that staircase join never touches a node in tables `context` and `doc` more than once (skipping, in fact, helps to avoid touching a significant number of nodes at all). This is in contrast to, *e.g.*, the $\mathcal{EE}/\mathcal{EA}$ join algorithms of [11] which repeatedly iterate over context and document in their inner loops.

6 Conclusions

The staircase join operator described in this paper exploits the tree properties encoded in the *pre*/*post* plane to optimize database-supported XPath evaluation. Knowledge of tree properties like subtree size and inclusion/disjointness of subtrees is available in this encoding at the cost of simple integer operations. We have shown that increased tree awareness can lead to significantly improved XPath performance.

To avoid cluttering the query optimizer with XML specifics, we propose to teach the RDBMS about tree properties by means of a local change to its kernel: the addition of a single join operator, the *staircase join*, encapsulating all "tree knowledge". The staircase join has been added to a main-memory RDBMS kernel. In this context, we could demonstrate that staircase join can be optimized well for in-memory operation.

Future Research

Ideally, we would like to experiment with a staircase join implementation in a commercial disk-based RDBMS, if possible. Secondly, in our experiments, we used documents of up to 1 GB. For even larger documents or large multi-document databases, one evidently needs to apply fragmentation strategies. An interesting strategy is to fragment by tag name. First experiments are encouraging: the execution time of $Q1$ could be brought down from 345 ms to 39 ms. This strategy should probably be combined with a partitioning-inspired one (see Section 3.2) when tag name fragments become too large. It should be obvious that fragmentation naturally leads to a parallel XPath execution strategy.

Further research goes in the direction of a cost model to be able to intelligently choose between name/node test pushdown and related XPath rewriting laws.

Acknowledgments

The authors would like to thank the Monet people at CWI (Amsterdam, The Netherlands) for their support and most useful feedback. Maurice van Keulen has been with the University of Konstanz as a DAAD INNOVATEC funded research fellow.

References

[1] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *The VLDB Journal*, 11(3), 2002.

[2] 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, November 2002. `http://www.w3.org/TR/xpath20/`.

[3] Scott Boag, Don Chamberlin, Fernandez Mary F., Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Technical Report W3C Working Draft, World Wide Web Consortium, November 2002. `http://www.w3.org/TR/xquery`.

[4] Peter A. Boncz. *Monet, a Next-Generation DBMS Kernel for Query-Intensive Applications.* PhD thesis, CWI Amsterdam, 2002.

[5] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proc. of the 28th VLDB Conference*, pages 263–274, Hong Kong, China, August 2002.

[6] Intel Corporation. *Intel Pentium® and Intel Xeon® Processor Optimization Reference Manual*, 2002.

[7] Tom Duff. `Netnews` posting. `http://www.jargon.net/jargonfile/d/Duffsdevice.html`, 1984.

[8] Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the 21st ACM SIGMOD Conference*, pages 109–120, Madison, Wisconsin, USA, June 2002. ACM Press.

[9] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proc. of the 19th ICDE Conference*, Bangalore, India, March 2003. IEEE Computer Society.

[10] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing Intervals Efficiently in Object-Relational Databases. In *Proc. of the 26th VLDB Conference*, pages 407–418, Cairo, Egypt, September 2000.

[11] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th VLDB Conference*, pages 361–370, Rome, Italy, September 2001.

[12] Stefan Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance.* PhD thesis, CWI Amsterdam, 2002.

[13] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. Symmetry in XPath. Technical Report PMS-FB-2001-16, Institute of Computer Science, University of Munich, Germany, October 2001.

[14] Kenneth A. Ross. Conjunctive Selection Conditions in Main Memory. In *Proc. of the 21st ACM Symposium on Principles of Database Systems (PODS)*, pages 109–120, Madison, Wisconsin, June 2002. ACM Press.

[15] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th VLDB Conference*, pages 974–985, Hong Kong, China, August 2002.

[16] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. of the 21st ACM Symposium on Principles of Database Systems (PODS)*, Madison, Wisconsin, June 2002. ACM Press.

[17] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the 20th ACM SIGMOD Conference*, pages 425–436, Santa Barbara, California, May 2001. ACM Press.