Tree Awareness for Relational DBMS Kernels: Staircase Join

Torsten Grust[1] and Maurice van Keulen[2]

[1] Department of Computer and Information Science, University of Konstanz, P.O. Box D188, 78457 Konstanz, Germany, Torsten.Grust@uni-konstanz.de
[2] Faculty of EEMCS, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands, m.vankeulen@utwente.nl

1 Introduction

Relational database management systems (RDBMSs) derive much of their efficiency from the versatility of their core data structure: *tables of tuples*. Such tables are simple enough to allow for an efficient representation on all levels of the memory hierarchy, yet sufficiently generic to host a wide range of data types. If one can devise mappings from a data type τ to tables and from operations on τ to relational queries, an RDBMS may be a premier implementation alternative. Temporal intervals, complex nested objects, and spatial data are sample instances for such types τ.

The key to efficiency of the relational approach is that the RDBMS is made *aware* of the specific properties of τ. Typically, such awareness can be implemented in the form of index structures (e.g., *R-trees* [7] efficiently encode the inclusion and overlap of spatial objects) or query operators (e.g., the *multi-predicate merge join* [11] exploits knowledge about containment of nested intervals).

This chapter applies this principle to the *tree* data type with the goal to turn RDBMSs into efficient XML and XPath processors [1]. The database system is supplied with a *relational* [8] XML document encoding, the *XPath accelerator* [5]. Encoded documents (1) are represented in relational tables, (2) can be efficiently indexed using index structures native to the RDBMS, namely B-trees, and (3) XPath queries may be mapped to SQL queries over these tables. The resulting purely relational XPath processor is efficient [5] and complete (supports all 13 XPath axes).

We will show that an enhanced level of *tree awareness*, however, can lead to a query speed-up by an order of magnitude. Tree awareness is injected into the database kernel in terms of the *staircase join* operator, which is tuned to exploit the knowledge that the RDBMS operates over tables encoding tree-shaped data. This is a local change to the database kernel: standard B-trees

suffice to support the evaluation of staircase join and the query optimizer may treat staircase join much like other native join operators.

2 Purely Relational XPath Processing

We will work with a relational XML encoding that is *not* inspired by the document tree structure per se – like, e.g., the *edge mapping* [4] – but by our primary goal to support XPath efficiently. The encoding of document trees is faithful nevertheless: properties like *document order*, *tag names*, *node types*, *text contents*, etc., are fully preserved such that the original XML document may be serialized from the relational tables alone.

Observe that for any element node of a given XML instance, the four XPath axes `preceding`, `descendant`, `ancestor`, and `following` *partition* the document into four regions. In Fig. 1, these regions are depicted for *context node* f: the XPath expression f`/following::node()`[3] evaluates to the node sequence (i, j), for example. Note that all 10 document nodes are covered by the four disjoint axis regions (plus the context node):

$$\{a \ldots j\} \quad = \quad \{f\} \cup \bigcup_{\alpha \in \{\texttt{preceding},\texttt{descendant},\texttt{ancestor},\texttt{following}\}} f/\alpha \ . \tag{1}$$

The *XPath accelerator* document encoding [5] preserves this region notion. The key idea is to design the encoding such that the nodes contained in an axis region can be retrieved by a relational query simple enough to be efficiently supported by relational index technology (in our case *B-trees*). Equation (1) guarantees that all document nodes are indeed represented in such an encoding.

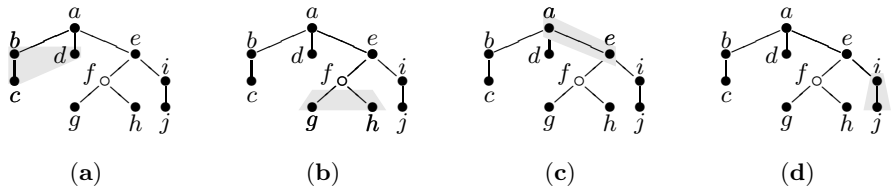


Fig. 1. XPath axes induce document regions: shaded nodes are reachable from context node f via a step along the (a) `preceding`, (b) `descendant`, (c) `ancestor`, (d) `following` axes. Leaf nodes denote either empty XML elements, attributes, text, comment, or processing instruction nodes; inner nodes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

The actual encoding maps each node v to its *preorder* and *postorder* traversal ranks in the document tree: $v \mapsto \langle pre(v), post(v) \rangle$. In a preorder traversal, a node v is visited and assigned its preorder rank $pre(v)$ before its children are

[3] In the sequel, we will abbreviate such XPath step expressions as f`/following`.

recursively traversed from left to right. Postorder traversal is defined dually: node v is assigned its postorder rank $post(v)$ after all its children have been traversed. For the XML document tree of Fig. 1, a preorder traversal enumerates the nodes in *document order* $(a, \ldots, j)$ while a postorder traversal enumerates $(c, b, d, g, h, f, j, i, e, a)$, so that we get $\langle pre(e), post(e)\rangle = \langle 4, 8\rangle$, for instance.

Figure 2 depicts the two-dimensional *pre/post plane* that results from encoding the XML instance of Fig. 1. Context node f, encoded as $\langle pre(f), post(f)\rangle = \langle 5, 5\rangle$, induces four *rectangular regions* in the *pre/post* plane, e.g., in the lower-left partition we find the nodes f/`preceding` $= (b, c, d)$. This characterization of the XPath axes is much more accessible for an RDBMS: an axis step can be evaluated in terms of a *rectangular region query* on the *pre/post* plane. Such queries are efficiently supported by concatenated $(pre, post)$ B-trees (or R-trees [5]).

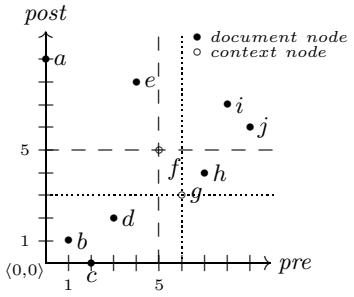


Fig. 2. *Pre/post* plane for the XML document of Fig. 1. Dashed and dotted lines indicate the document regions as seen from context nodes f (− −) and g (·····), respectively

The further XPath axes, like, e.g., `following-sibling` or `ancestor-or-self`, determine specific supersets or subsets of the node sets computed by the four partitioning axes. These are easily characterized if we additionally maintain *parent node* information for each node, i.e., use $v \mapsto \langle pre(v), post(v), pre(parent(v))\rangle$ as the encoding for node v. We will focus on the four partitioning axes in the following.

Note that all nodes are alike in the XPath accelerator encoding: given an arbitrary context node v, e.g., computed by a prior XPath axis step or XQuery expression, we retrieve $\langle pre(v), post(v)\rangle$ and then access the nodes in the corresponding axis region. Unlike related approaches [2], the XPath accelerator has no bias towards the document root element. Please refer to [5, 6] for an in-depth explanation of the XPath accelerator.

2.1 SQL-based XPath Evaluation

Inside the relational database system, the encoded XML document tree, i.e., the *pre/post* plane, is represented as a table `doc` with schema *pre*|*post*|*type*. Each tuple encodes a single node (with field *type* discriminating element, attribute, text, comment, processing instruction node types). Since *pre* is unique – and thus may serve as *node identity* as required by the W3C XQuery and XPath data model [3] – additional node information is assumed to be hosted in

separate tables using *pre* as a foreign key.[4] A SAX-based document loader [9] can populate the doc table using a single sequential scan over the XML input [5].

The evaluation of an XPath path expression $p = s_1/s_2/\cdots/s_n$ leads to a series of n region queries where the node sequence output by step s_i is the context node sequence for the subsequent step s_{i+1}. The context node sequence for step s_1 is held in table context (if p is an *absolute* path, i.e., $p = /s_1/\cdots$, context holds a single tuple: the encoding of the document root node). XPath requires the resulting node sequence to be *duplicate free* as well as being sorted in *document order* [1]. These inherently set-oriented, or rather *sequence-oriented*, XPath semantics are implementable in plain SQL (Fig. 3).

```
1  SELECT DISTINCT v_n.*
2    FROM context c, doc v_1, ..., doc v_n
3   WHERE axis(s_1, c, v_1) AND axis(s_2, v_1, v_2) AND ··· AND axis(s_n, v_{n-1}, v_n)
4  ORDER BY v_n.pre ASC
```

$$
\begin{aligned}
axis(\texttt{preceding}, v, v') &\equiv v'.pre < v.pre \ \texttt{AND}\ v'.post < v.post\\
axis(\texttt{descendant}, v, v') &\equiv v'.pre > v.pre \ \texttt{AND}\ v'.post < v.post\\
axis(\texttt{ancestor}, v, v') &\equiv v'.pre < v.pre \ \texttt{AND}\ v'.post > v.post\\
axis(\texttt{following}, v, v') &\equiv v'.pre > v.pre \ \texttt{AND}\ v'.post > v.post
\end{aligned}
$$

Fig. 3. Translating the XPath path expression $s_1/s_2/\cdots/s_n$ (with context context) into an SQL query over the document encoding table doc

Note that we focus on the XPath core, namely location steps, here. Function $axis(\cdot)$, however, is easily adapted to implement further XPath concepts, like node tests, e.g., with XPath axis α and node kind $\kappa \in \{\texttt{text()}, \texttt{comment()}, \ldots\}$:

$$axis(\alpha\texttt{::}\kappa, v, v') \equiv axis(\alpha, v, v') \ \texttt{AND}\ v'.type = \kappa \ .$$

Finally, the existential semantics of XPath predicates are naturally expressed by a simple exchange of correlation variables in the translation scheme of Fig. 3. The XPath expression $s_1\texttt{[}s_2\texttt{]}/s_3$ is evaluated by the RDBMS via the SQL query shown in Fig. 4.

```
1  SELECT DISTINCT v_3.*
2    FROM context c, doc v_1, doc v_2, doc v_3
3   WHERE axis(s_1, c, v_1) AND axis(s_2, v_1, v_2) AND axis(s_3, v_1, v_3)
4  ORDER BY v_3.pre ASC .
```

Fig. 4. SQL equivalent for the XPath expression $s_1\texttt{[}s_2\texttt{]}/s_3$ (note the exchange of v_1 for v_2 in $axis(s_3, v_1, v_3)$, line 3).

[4] In this chapter, we will not discuss the many possible table layout variations (in-line tag names or CDATA contents, partition by tag name, etc.) for doc.

2.2 Lack of Tree Awareness in Relational DBMS

The structure of the generated SQL queries – a flat self-join of the doc table using a conjunctive join predicate – is simple. An analysis of the actual query plans chosen by the optimizer of IBM DB2 V7.1 shows that the system can cope quite well with this type of query. Figure 5 depicts the situation for a two-step query s_1/s_2 originating in context sequence context.

The RDBMS maintains a B-tree over concatenated $(pre, post)$ keys. The index is used to scan the inner (right) doc table join inputs in *pre*-sorted order. The context is, if necessary, sorted by the preorder rank *pre* as well. Both joins may thus be implemented by *merge joins*. The actual region query evaluation happens in the two inner join inputs: the predicates on *pre* act as index range scan delimiters while the conditions on *post* are fully sargable [10] and thus evaluated during the B-tree index scan as well. The joins are actually right semijoins, producing their output in *pre*-sorted order (which matches the request for a result sequence sorted in document order in line 4 of the SQL query).

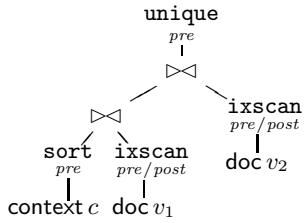


Fig. 5. Query plan

As reasonable as this query plan might appear, the RDBMS treats table doc (and context) like any other relational table and remains ignorant of tree-specific relationships between $pre(v)$ and $post(v)$ other than that both ranks are paired in a tuple in table doc. The system thus gives away significant optimization opportunities.

To some extent, however, we are able to make up for this lack of tree awareness at the SQL level. As an example, assume that we are to take a `descendant` step from context node v (Fig. 6). It is sufficient to scan the $(pre, post)$ B-tree in the range from $pre(v)$ to $pre(v')$ since v' is the rightmost leaf in the subtree below v and thus has maximum preorder rank. Since the *pre*-range $pre(v)$–$pre(v')$ contains exactly the nodes in the `descendant` axis of v, we have[5]

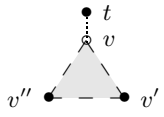


Fig. 6. Nodes with minimum *post* (v'') and maximum *pre* (v') ranks in the subtree below v

$$pre(v') = pre(v) + |v/\texttt{descendant}| \quad . \tag{2}$$

Additionally, for any node v in a tree t we have that

$$|v/\texttt{descendant}| = post(v) - pre(v) + \underbrace{level(v)}_{\leqslant h} \tag{3}$$

where $level(v)$ denotes the length of the path from t's root to v which is obviously bound by h, the overall *height* of t.[6] Equations (2) and (3) provide

[5] We use $|s|$ to denote the cardinality of set s.

[6] The system can compute h at document loading time. For typical real-world XML instances, we have $h \leqslant 10$.

us with a characterization of $pre(v')$ expressed exclusively in terms of the current context node v:

$$pre(v') \leqslant post(v) + h \ . \tag{4}$$

A dual argument applies to leaf v'', the node with minimum postorder rank below context node v (Fig. 6). Taken together, we can use these observations to further delimit the B-tree index range scans to evaluate `descendant` axis steps:

$$\begin{aligned} axis(\texttt{descendant}, v, v') \ \equiv \ & v'.pre > v.pre \ \texttt{AND} \ v'.pre \leq v.post + h \ \texttt{AND} \\ & v'.post \geq v.pre + h \ \texttt{AND} \ v'.post < v.post \ . \end{aligned}$$

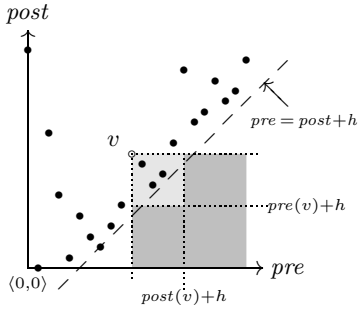


Fig. 7. Original (dark) and shrunk (light) *pre* and *post* scan ranges for a `descendant` step to be taken from v

Note that the index range scan is now delimited by the actual size of the context nodes' subtrees – modulo a small misestimation of maximally h which is insignificant in multi-million node documents – and independent of the document size. The benefit of using these shrunk `descendant` axis regions is substantial, as Fig. 7 illustrates for a small XML instance. In [5], a speed-up of up to three orders of magnitude has been observed for 100 MB XML document trees.

Nevertheless, as we will see in the upcoming section, the index scans and joins in the query plan of Fig. 5 still perform a significant amount of wasted work, especially for large context sequences. Being uninformed about the fact that the `doc` tables encodes tree-shaped data, the index scans repeatedly re-read regions of the *pre*/*post* plane only to generate duplicate nodes. This, in turn, violates XPath semantics such that a rather costly duplicate elimination phase (the `unique` operator in Fig. 5) at the top of the plan is required.

Real tree awareness, however, would enable the RDBMS to improve XPath processing in important ways: (1) since the node distribution in the *pre*/*post* plane is not arbitrary, the `ixscan`s could actually *skip* large portions of the B-tree scans, and (2) the context sequence induces a partitioning of the plane that the system can use to fully avoid duplicates.

The necessary tree knowledge *is* present in the *pre*/*post* plane – and actually available at the cost of simple integer operations like $+, <$ as we will now see – but remains inaccessible for the RDBMS unless it can be made explicit at the SQL level (like the `descendant` window optimization above).

3 Encapsulating Tree Awareness in the Staircase Join

To make the notion of *tree awareness* more explicit, we first analyze some properties of trees and how these are reflected in the *pre/post*-plane encoding. We introduce three techniques, *pruning*, *partitioning*, and *skipping*, that exploit these properties. The section concludes with the algorithm for the *staircase join*, a new join operator that incorporates the three techniques.

3.1 Pruning

In XPath, an axis step is generally evaluated on an *entire sequence* of context nodes [1]. This leads to duplication of work if the *pre/post* plane regions associated with the step are independently evaluated for each context node.

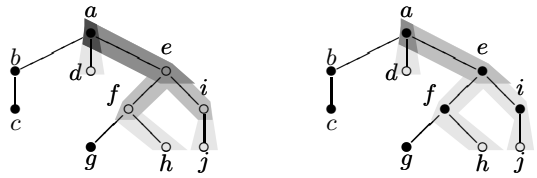
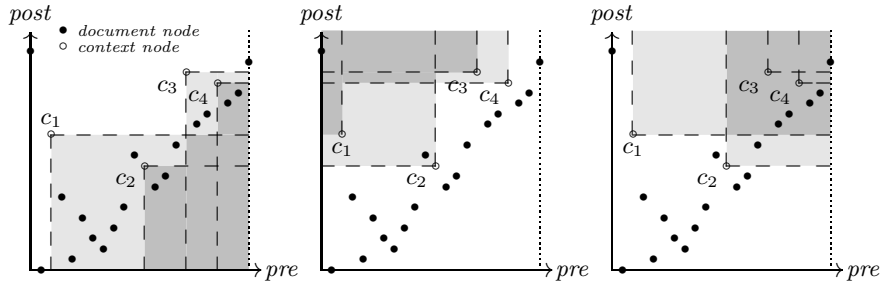


Fig. 8. (a) Intersection and inclusion of the `ancestor-or-self` paths of a context node sequence. (b) The pruned context node sequence covers the same `ancestor-or-self` region and produces less duplicates (3 rather than 11)

Figure 8 (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 a costly duplicate elimination phase. 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) (Fig. 8 (b)). Such opportunities for the simplification of the context node sequence arise for all axes.

Figure 9 depicts the scenario 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. The process of identifying the context nodes at the cover's boundary is referred to as *pruning* and is easily implemented involving a simple postorder rank comparison (Fig. 14).



(**a**) `descendant` axis (**b**) `ancestor` axis (**c**) `following` axis

Fig. 9. Overlapping regions (context nodes c_i)

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 Fig. 10. The context establishes a boundary in the *pre*/*post* plane that resembles a *staircase*.

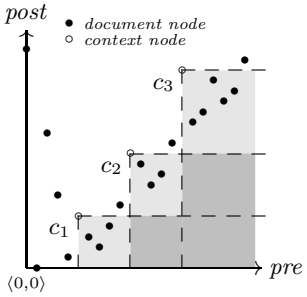
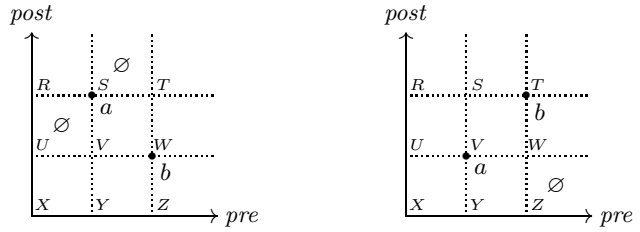


Fig. 10. Context pruning produces proper staircases

Observe in Fig. 10 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 Fig. 11). 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 Fig. 10 correspond to such Z regions.

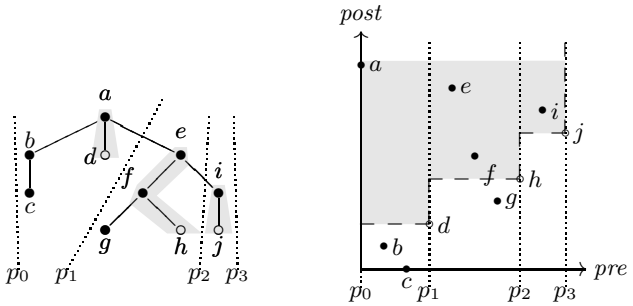
A similar *empty region analysis* can be done for all XPath axes. The consequences for the `preceding` and `following` axes are more profound. After pruning for, e.g., the `following` axis, the remaining context nodes relate to each other on the `ancestor`/`descendant` axis. In Fig. 11 (a), we see that 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 axis step



(**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

Fig. 11. Empty regions in the *pre*/*post* plane



(**a**) (**b**)

Fig. 12. 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

evaluation degenerates to a single region query. We will therefore focus on the `ancestor` and `descendant` axes in the following.

3.2 Partitioning

While pruning leads to a significant reduction of duplicate work, Fig. 8 (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 (Fig. 12 (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 (Fig. 12 (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 that the partitioned *pre*/*post* plane naturally leads to a parallel XPath execution strategy.)

3.3 Skipping

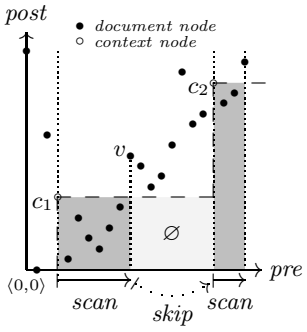


Fig. 13. Skipping technique for `descendant` axis

The empty region analysis explained in Sect. 3.1 offers another kind of optimization, which we refer to as *skipping*. Figure 13 illustrates this for the XPath axis step (c_1, c_2)`/descendant`. An axis step can be evaluated by scanning the *pre*/*post* plane from left to right and partition by partition 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 the 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 Fig. 13 is a region of type Z in Fig. 11 (b).

This observation can be used to terminate the scan early which effectively means that the portion of the scan between $pre(v)$ and the successive context node $pre(c_2)$ is *skipped*.

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 $|\mathsf{result}| + |\mathsf{context}|$ nodes in the *pre*/*post* plane, a number independent of the document size.

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 (3) 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 $post(v) - pre(v)$.

3.4 Staircase Join Algorithm

The techniques of pruning, partitioning, and skipping are unavailable to an RDBMS that is not "tree aware". 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" of pruning, partitioning, and skipping present in the *pre*/*post* plane. On the outside, it behaves to the query optimizer in many ways as an ordinary join, for example, by admitting selection pushdown.

The approach to evaluating a staircase join between a document and a context node sequence 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 Fig. 14). Along the way, encountered context nodes are pruned when possible. Furthermore, portions of a partition that are guaranteed not to contain any result nodes are skipped. 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.

```
staircasejoin_desc (doc : TABLE (pre,post), context : TABLE (pre,post)) ≡
  result ← NEW TABLE (pre, post);
  /* partition c_from ... c_to */
  c_from ← FIRST NODE IN context;
  WHILE (c_to ← NEXT NODE IN context) DO
    IF c_to.post < c_from.post THEN
      /* prune */
    ELSE
      scanpartition_desc (c_from.pre + 1, c_to.pre − 1, c_from.post);
      c_from ← c_to;
  n ← LAST NODE IN doc;
  scanpartition_desc (c_from.pre + 1, n.pre, c_from.post);
  RETURN result;
scanpartition_desc (pre_from, pre_to, post) ≡
  FOR i FROM pre_from TO pre_to DO
    IF doc[i].post < post THEN
      APPEND doc[i] TO result;
    ELSE
      BREAK;    /* skip */
```

Fig. 14. Staircase join algorithm (`descendant` axis, `ancestor` analogous)

This 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 scans a fraction of table doc with a size smaller than |result| + |context|,
(4) it never delivers duplicate nodes, and

(5) result nodes are produced in document order, so no post-processing is needed to comply with the XPath semantics.

4 Query Planning with Staircase Join

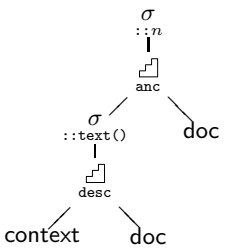


Fig. 15. Plan for two-step path query

The evaluation of an XPath path expression $p = s_1/s_2/\cdots/s_n$ leads to a series of n region queries where the node sequence output by step s_i is the context node sequence for the subsequent step s_{i+1} (see Sect. 2.1). One axis step s_i encompasses a location step and possibly a node test. The corresponding query plan, hence, consists of a staircase join for the location step and a subsequent selection for the node test. Figure 15 shows a possible query plan for the example query `context/descendant::text()/ancestor::`n ($\underset{\texttt{desc}}{\text{⌐}}$ and $\underset{\texttt{anc}}{\text{⌐}}$ depict the `descendant` and `ancestor` variants of the staircase join, respectively).

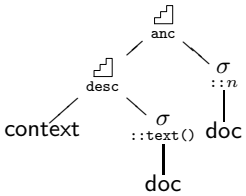


Fig. 16. Query plan with node test pushed down

There are, however, alternative query plans. Staircase join, like ordinary joins, allows for *selection pushdown*, or rather node test pushdown: for any location step α and node test κ

$$\sigma_{::\kappa}(\mathsf{context} \underset{\alpha}{\text{⌐}} \mathsf{doc}) \;=\; \mathsf{context} \underset{\alpha}{\text{⌐}} (\sigma_{::\kappa}(\mathsf{doc})) \;\; .$$

Figure 16 shows a query plan for the example query where both node tests have been pushed down.

Observe that in the second query plan, the node test is performed on the *entire* document instead of just the result of the location step. An RDBMS already keeps *statistics* about table sizes, selectivity, and so on. These can be used by the query optimizer in the ordinary way to decide whether or not the node test pushdown makes sense. *Physical database design* does not require exceptional treatment either. For example, in a setting where applications mainly perform qualified name tests (i.e., few '`::*`' name tests), it is beneficial to fragment table doc by tag name. A pushed down name test $\sigma_{::n}(\mathsf{doc})$ can then be evaluated by taking the appropriate fragment without the need for any data processing.

The addition of staircase join to an existing RDBMS kernel and its query optimizer is, by design, a local change to the database system. A standard B-tree index suffices to realize the "tree knowledge" encapsulated in staircase join. Skipping, as introduced in Sect. 3.3, is efficiently implemented by following the *pre*-ordered chain of linked B-tree leaves, for example.

We have found staircase join to also operate efficiently on higher levels of the memory hierarchy, i.e., in a main-memory database system. For queries like the above example, the staircase join enhanced system processed 1 GB

XML documents in less than 1/2 second on a standard single processor host [6].

5 Conclusions

The approach toward efficient XPath evaluation described in this paper is based on a *relational* document encoding, the *XPath accelerator*. A preorder plus postorder node ranking scheme is used to encode the tree structure of an XML document. In this scheme, XPath axis steps are evaluated via joins over simple integer range predicates expressible in SQL. In this way, the XPath accelerator naturally exploits standard RDBMS query processing and indexing technology.

We have shown that an enhanced level of *tree awareness* can lead to a significant speed-up. This can be obtained with only a local change to the RDBMS kernel: the addition of the *staircase join* operator. This operator encapsulates XML document tree knowledge by means of incorporating the described techniques of *pruning*, *partitioning*, and *skipping* in its underlying algorithm. The new join operator requires no exceptional treatment: staircase join affects physical database design and query optimization much like traditional relational join operators.

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, November 2002. `http://www.w3.org/TR/xpath20/`.
2. Brain F. Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proc. of the 27th Int'l Conference on Very Large Data Bases (VLDB)*, pages 341–360, Rome, Italy, September 2001.
3. Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XML Query Data Model. Technical Report W3C Working Draft, World Wide Web Consortium, November 2002. `http://www.w3.org/TR/query-datamodel`.
4. Daniela Florescu and Donald Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical Report 3680, INRIA, Rocquencourt, France, May 1999.
5. 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, June 2002.
6. 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 Data Bases (VLDB)*, Berlin, Germany, September 2003.
7. Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD 1984, Proc. of Annual Meeting*, pages 47–57, Boston, Massachusetts, June 1984. ACM Press.
8. Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing Intervals Efficiently in Object-Relational Databases. In *Proc. of the 26th Int'l Conference on Very Large Databases (VLDB)*, pages 407–418, Cairo, Egypt, September 2000.
9. SAX (Simple API for XML). `http://sax.sourceforge.net/`.
10. Patricia G. Selinger, Morton M. Astrahan, Donald M. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 23–34, Boston, Massachusetts, USA, 1979.
11. Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Sys-

tems. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 425–436, Santa Barbara, California, May 2001. ACM Press.