Enhancing the Tree Awareness of a Relational DBMS

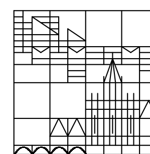
Adding Staircase Join to PostgreSQL

Master Thesis

Sabine Mayer
Brüelstr. 7, 78462 Konstanz
mayers@inf.uni-konstanz.de

February 2004

Universität Konstanz
Fachbereich Informatik und
Informationswissenschaft



Contents

1 Introduction **1**

2 Tree Awareness for an RDBMS **3**
2.1 XPath Expressions . . . 3
2.2 The XPath Accelerator . . . 4
2.2.1 Region Queries . . . 5
2.2.2 Translation into SQL Syntax . . . 6
2.3 The Staircase Join . . . 7
2.3.1 Context Pruning . . . 8
2.3.2 Partitioning and the Staircase Join Algorithms . . . 11
2.3.3 Skipping . . . 14

3 Implementation of the Staircase Join in PostgreSQL **17**
3.1 Query Processing in the PostgreSQL Database Engine . . . 17
3.1.1 The Parser . . . 18
3.1.2 The Rewriter . . . 18
3.1.3 The Planner/Optimizer . . . 18
3.1.4 The Executor . . . 19
3.2 Restrictions in a Tree-Aware PostgreSQL Instance . . . 20
3.2.1 Table-Specific Restrictions . . . 20
3.2.2 Query-Specific Restrictions . . . 20
3.2.3 Availability of Indices . . . 21
3.3 Implementation Steps in the Parser . . . 21
3.4 Implementation Steps in the Planner/Optimizer . . . 21
3.4.1 Clause Preparation . . . 21
3.4.2 Preparation of Access Paths . . . 22
3.4.3 Marking the Context Set . . . 23
3.4.4 Dynamic Programming . . . 24
3.4.5 Conversion into an Execution Plan . . . 26
3.4.6 Distinctness and Sort Order of the Final Result . . . 28
3.5 Implementation Steps in the Executor . . . 28
3.5.1 Initialization of Staircase Join Execution . . . 29
3.5.2 Pipelining . . . 29
3.5.3 The Staircase Join's Execution Module . . . 30
3.5.4 Context Pruning, Partitioning, and Skipping . . . 30
3.5.5 Completing Staircase Join Execution . . . 37
3.5.6 Further Implementation Considerations . . . 37

4 Performance Tests **39**
4.1 The Test Environment . . . 39
4.1.1 Creation of Test Documents . . . 39
4.1.2 Loading the XML Documents . . . 39

4.1.3 Test Queries 40
4.1.4 Execution Plans 41
4.2 Evaluation of Test Results 42
4.2.1 Disk Page Loads and Buffer Hits 43
4.2.2 Execution Times 48

5 Conclusion and Outlook **51**
5.1 The PostgreSQL Backend 51
5.2 The Test Results 53
5.3 Outlook 54

A Data Structures in the Planner/Optimizer **55**
A.1 The `Query` Structure 55
A.2 The `RelOptInfo` Structure 55
A.3 The `Path` Structure 56
A.4 The `PathKeyItem` Structure 56
A.5 The `RestrictInfo` Structure 56
A.6 The `JoinInfo` Structure 57
A.7 The `Plan` Structure 57

B New Source Code in the Planner/Optimizer **59**
B.1 Clause Preparation 59
B.2 Dynamic Programming 60
B.3 Conversion into an Execution Plan 60

C Data Structures in the Executor **63**
C.1 The `TupleTableSlot` Structure 63
C.2 The `EState` Structure 63
C.3 The `JoinState` Structure 63
C.4 The `ExprContext` Structure 63

D New Source Code in the Executor **65**
D.1 Initialization of Staircase Join Execution 65
D.2 The Staircase Join's Execution Module 65
D.3 Completing Staircase Join Execution 66

E DTD of the XML Test Documents **67**

Abstract

Given a suitable encoding, any relational DBMS is able to answer queries on tree-structured data. However, conventional relational databases are generally not (made) aware of the underlying tree structure and thus fail to make full use of the encoded information.

The *staircase join* is a new join algorithm intended to *enhance the tree awareness* of a relational DBMS. It was developed to speed up the SQL-based evaluation of XPath expressions. The algorithm encapsulates tree-specific knowledge and relies on the data provided by the *XPath accelerator*, an encoding which maps information about the tree-shaped structure of an XML document to a relational database table.

This thesis shows that it is possible to incorporate the staircase join into a conventional RDBMS, namely the open-source RDBMS PostgreSQL. The implementation involved local changes to three out of four query processing stages in the PostgreSQL backend: the parser, the planner/optimizer, and the executor.

The performance tests subsequently carried out in the tree-aware PostgreSQL instance confirmed that the staircase join leads to a substantial query speed-up. In comparison to the native join algorithm which is chosen by the original PostgreSQL database to evaluate SQL-based XPath expressions, the staircase join produced an improvement up to several orders of magnitude. Thus, the tests have shown that, in conjunction with a suitable cost model, the staircase join can turn a relational database system into an efficient XML query processing solution.

Chapter 1

Introduction

The success of XML (Extensible Markup Language) has caused a great demand for solutions that can efficiently manage large amounts of tree-structured data.

The members of the *Pathfinder* working group, situated at the Universities of Konstanz (Germany) and Twente (The Netherlands), rely on relational database management systems to store XML data and query them on the basis of XPath expressions. They have shown that a suitable encoding, which maps the XML document tree to a relational table, is basically enough to put this approach into practice. However, since relational databases generally lack awareness of the underlying tree structure, they fail to make full use of the encoded information. To solve this problem, the *Pathfinder* members have developed a new join algorithm, the *staircase join* [GvKT03, GvK03], which *enhances the tree awareness* of a relational database. It was designed to speed up the SQL-based evaluation of XPath expressions. The algorithm incorporates tree-specific knowledge and is based on the *XPath accelerator* [Gru02, GvKT04], a relational XML document encoding which provides inherent support for the evaluation of all 13 XPath location steps. Promising results have already been obtained in the main-memory database system *Monet* [Bon02].

In the course of this thesis, the staircase join algorithm was incorporated into the open-source, relational DBMS PostgreSQL and its performance behavior was examined. The test results were then compared to the execution times of the native join algorithm which was chosen by the unmodified PostgreSQL database to answer SQL-based XPath expressions. The measured performance speed-up confirmed the positive results obtained in Monet.

The thesis proceeds as follows. Chapter 2 provides the theoretical background for the implementation of the staircase join in PostgreSQL. It is dedicated to the concepts behind the XPath accelerator and the staircase join itself. Chapter 3 provides an insight into the actual implementation. It starts off with an overview of the query processing stages in the PostgreSQL backend and proceeds with a few table- and query-specific restrictions that facilitate and support the upcoming implementation tasks. The following three sections of the chapter are dedicated to the implementation steps that were undertaken in the parser, planner/optimizer, and executor module of PostgreSQL, respectively. Chapter 4 describes the performance tests that were carried out and provides detailed information about the obtained test results. Finally, Chapter 5 gives a general conclusion and provides an outlook to potential future implementation work.

The CD included in this thesis contains the staircase join-enhanced version of PostgreSQL 7.3.3. To facilitate the search for source code that was incorporated into the database backend in the course of the implementation, the newly added lines of code were put between two comments of the form:

```
/* BOInsert: ... */
      ⋮
/* EOInsert */.
```

The original version of PostgreSQL 7.3.3 can be downloaded from the PostgreSQL web site at `http://www.postgresql.org`.

Chapter 2

Tree Awareness for an RDBMS

This section provides a theoretical insight into the research areas explored by the *Pathfinder* working group to enhance the tree awareness of relational databases. It starts off with a general overview of XPath evaluation and proceeds with a detailed description of the *XPath accelerator* and the *staircase join*. Further background information on both concepts are available in [Gru02], [GvKT03], [GvK03], and [GvKT04]. The web site of the *Pathfinder* working group can be found at `http://www.inf.uni-konstanz.de/dbis/research/pathfinder/`.

2.1 XPath Expressions

The elements and attributes in an XML document are organized in a tree-structured node hierarchy and XPath [BBC+03] provides a total of 13 axes to address a specified subset of those document nodes, a so-called *document region*. An XPath expression consists of at least one such *axis* or *location step* α and starts at a given context node v.[1] Table 2.1 describes the regions that can be addressed in an XML document by a single location step v/α.

If the XPath expression consists of several location steps, they are syntactically separated by slashes (/), for example,

$$v/\alpha_0/\alpha_1/\alpha_2/.../\alpha_n.$$

The evaluation of such an expression is directed from left to right, i.e., the result of location step α_i is used as context set for the evaluation of the subsequent step α_{i+1}. Each location step in the expression must be evaluated once per node in its context set. The result of the step is the union of the obtained document regions, the nodes being sorted in document order. The result of the very last location step α_n is the overall result of the XPath expression.

Consider the example XML document displayed in Figure 2.1. Given the context node b and the XPath expression $b/\texttt{following}/\texttt{descendant}$, we obtain the following mode of evaluation:

$$b/\texttt{following}/\texttt{descendant} \quad \equiv \quad (d,e,f,g,h,i,j)/\texttt{descendant} \quad \equiv \quad (f,g,h,i,j).$$

[1] Typically, there is one context node, which often, but not necessarily, corresponds to the root node of the XML document. However, there may also be a set of context nodes, for example, when the XPath expression is embedded in an XQuery [BCF+03] expression.

axis α	**result nodes**
`child`	child nodes of v
`descendant`	closure of `child`
`descendant-or-self`	like `descendant`, plus v
`parent`	parent node of v
`ancestor`	closure of `parent`
`ancestor-or-self`	like `ancestor`, plus v
`following`	nodes following v in the tree (excluding descendants)
`preceding`	nodes preceding v in the tree (excluding ancestors)
`following-sibling`	like `following`, same parent as v
`preceding-sibling`	like `preceding`, same parent as v
`attribute`	attribute nodes owned by v
`self`	v
`namespace`	namespace nodes owned by v

Table 2.1: The document regions established by location step v/α. v is the context node.

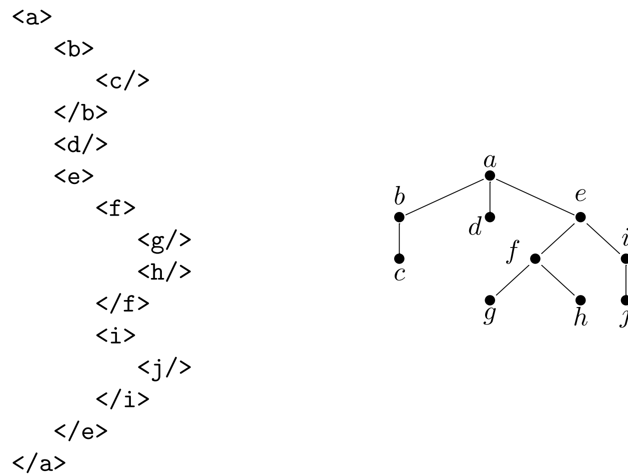


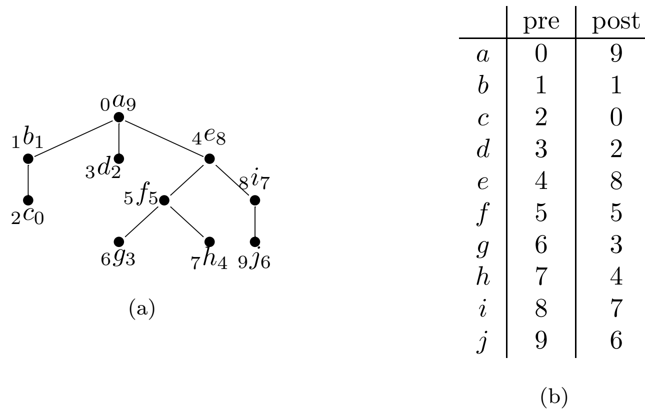
Figure 2.1: A simple XML document and its tree representation.

2.2 The XPath Accelerator

Before an XPath expression can be evaluated in a relational database, the queried XML document must be mapped to a database table. However, the mapping must not only take the explicitly contained information into account (e.g. node types, tag and attribute names, and text content), but must also preserve knowledge about the structural relations between the XML nodes.

The XPath accelerator is an encoding which maps information about the XML node hierarchy to a relational table and which is geared towards the efficient evaluation of XPath expressions. The encoding assigns a unique pair of numeric values, the *preorder* and *postorder traversal rank*, to each XML node. In a sequential read of the XML document, the preorder rank is assigned to a node when its start tag is visited, while the postorder rank is assigned when its end tag is visited. In the tree representation, this means that a node obtains its preorder rank *before* all its children are traversed from left to right and its postorder rank *after* all its children have been traversed. Figure 2.2 (a) shows a simple XML document tree with pre and post values assigned to the nodes. The relation in which the encoded values

are stored will be referred to as the *document table* in the following. An example of such a table is illustrated in Figure 2.2 (b).



	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

(b)

Figure 2.2: A simple XML document tree with preorder rank (left of tag name) and postorder rank (right of tag name) assigned to each node (a) and the respective document table (b).

2.2.1 Region Queries

If the encoded values are used to depict the nodes of the XML document in the *pre/post plane* — a two-dimensional graph in which the pre value is mapped to the x and the post value to the y axis —, it becomes apparent that the XPath accelerator has preserved an important property which is illustrated in Figure 2.3 (a): any context node v divides the XML document into four disjoint regions, which correspond to the result of the XPath location steps v/`preceding`, v/`ancestor`, v/`following`, and v/`descendant`, respectively. Figure 2.3 (b) illustrates that the same behavior can be observed with respect to the corresponding document regions in the tree representation of the XML document. In both cases, the union of these four regions covers all document nodes except the context node v.

The boundaries of the `preceding`, `ancestor`, `following`, and `descendant` regions — and thus the nodes contained within them — can be determined for any arbitrary context node v with the so-called *region queries*. Thanks to the XPath accelerator encoding, they amount to a pair of simple integer comparisons between the pre and post value of context node v and the pre and post values of the nodes in the XML document:

– The `preceding` nodes of context node v have a lower preorder and a lower postorder rank than v.

– The `following` nodes of context node v have a higher preorder and a higher postorder rank than v.

– The `descendant` nodes of context node v have a higher preorder and a lower postorder rank than v.

– The `ancestor` nodes of context node v have a lower preorder and a higher postorder rank than v.

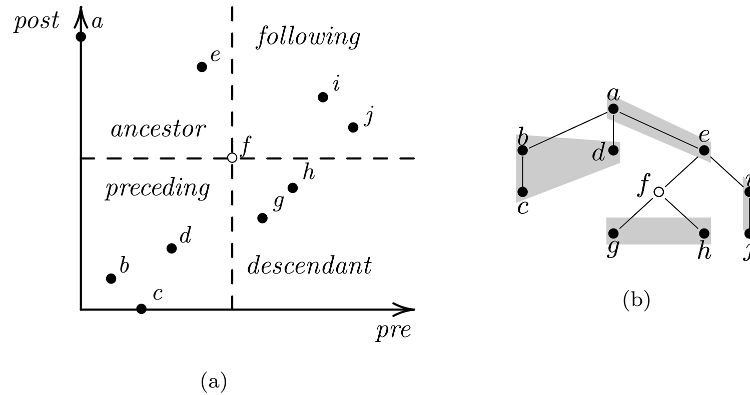


Figure 2.3: The `preceding`, `ancestor`, `following`, and `descendant` regions in the pre/post plane (a) and (clockwise, starting at the leftmost gray area) in the tree representation of an XML document (b). Context node is f.

2.2.2 Translation into SQL Syntax

Based on the knowledge encoded by the XPath accelerator and the region queries, we can now start to translate XPath expressions into SQL queries. The applied translation scheme closely follows the evaluation principles described for native XPath expressions in Section 2.1. The series of location steps is converted into a series of joins where each join links the result of the previous join (the *current context set*) to an instance of the XML document, more precisely the document table doc_i. The starting point of the first join or location step is the explicitly specified context set cs. We assume that it is also available as a relational table and contains the pre and post value of the initial context node(s). The XPath expression $v/\alpha_0/\alpha_1/\alpha_2/.../\alpha_n$ will thus be translated into the join sequence: $cs \bowtie doc_1 \bowtie doc_2 \bowtie ... \bowtie doc_n$.

The *join clauses* for each location step correspond to the respective pair of region queries established in Section 2.2.1. For any two joined relations r_1 and r_2 in the join sequence, with r_1 representing the relation that links the current join to the result of the previous one and r_2 representing a new instance of the document table, the following join clauses can be derived in dependence on the evaluated XPath axis:

$$
\begin{aligned}
axis(\texttt{preceding},\ r_1,\ r_2) &\equiv r_1.pre > r_2.pre \ \texttt{AND}\ r_1.post > r_2.post \\
axis(\texttt{following},\ r_1,\ r_2) &\equiv r_1.pre < r_2.pre \ \texttt{AND}\ r_1.post < r_2.post \\
axis(\texttt{descendant},\ r_1,\ r_2) &\equiv r_1.pre < r_2.pre \ \texttt{AND}\ r_1.post > r_2.post \\
axis(\texttt{ancestor},\ r_1,\ r_2) &\equiv r_1.pre > r_2.pre \ \texttt{AND}\ r_1.post < r_2.post.
\end{aligned}
$$

Since the XPath Working Draft of the W3C [BBC+03] demands that the result of an XPath expression is *duplicate-free* and *sorted in document order*, i.e. on the pre value, the SQL keywords `DISTINCT` and `ORDER BY` have to be present in the query. Given the example XPath expression cs/`following`/`descendant`, the SQL equivalent thus reads as follows:

```
SELECT DISTINCT doc2.*
  FROM context cs, document doc1, document doc2
 WHERE cs.pre < doc1.pre AND cs.post < doc1.post        -- following
   AND doc1.pre < doc2.pre AND doc1.post > doc2.post    -- descendant
 ORDER BY doc2.pre;
```

The four axes listed above will constitute the main focus of this thesis. This is because existing relational databases generally do not provide efficient support for their evaluation. The remaining axes can be divided into three categories:

- The `descendant-or-self` and `ancestor-or-self` axes are implicitly covered by the concepts presented in this thesis. Support for them can be achieved by a slight modification of the `descendant` and `ancestor` region queries. Each of their operators must be equipped with an additional equals sign to include the context nodes into the result as well.

- The `child`, `parent`, `attribute`, `self`, and `namespace` axes can be efficiently processed by existing evaluation techniques. However, this requires the inclusion of additional information into the document table. For an efficient evaluation of the `child` and `parent` axes, we can either add the preorder rank of each node's parent or the tree level of each node to the document table. The `attribute` axis requires the table to additionally store information about the node type[2], while the `namespace` axis can be evaluated, if we include a further namespace attribute into the table. A column reserved for the tag or attribute name of the nodes will allow for name tests in general.

- The `following-sibling` and `preceding-sibling` axes require a combination of new and existing database techniques.

2.3 The Staircase Join

The XPath accelerator has prepared the ground for storing and querying XML data in a relational database. However, without any tree-specific knowledge, the execution of an SQL-based XPath query in a conventional RDBMS may produce a lot of duplicate work. The reason for this overhead is illustrated in Figure 2.4. It shows that the evaluation of a location step amounts to a per-context-node computation of the respective document region (e.g. the three gray regions in Figure 2.4 (a) originating at context nodes d, e, and h). Applied to a relational database, this means that the region queries (the join clauses) must be evaluated once per node in the context set, which in turn results in the demand for a repeated scan of the document table. Apart from that, Figure 2.4 shows, that there may be a considerable overlap of the computed document regions. It is an indication that parts of the evaluation work are actually redundant, which is also reflected in the presence of duplicate nodes in the result.

The staircase join relies on three techniques (*pruning*, *partitioning*, and *skipping*) to avoid this kind of overhead. Given a pre-sorted context set and document table, it makes sure that:

- the evaluation of an XPath location step requires only a single sequential scan of both tables (in fact, some nodes may not have to be accessed at all) and that

- the result of each location step is duplicate-free and sorted on the pre value, i.e. sorted in document order. (Note that this makes the result comply with the XPath specifications without any further processing steps.)

All techniques rely on tree-specific knowledge of which a conventional database is not (made) aware.

[2] We assume that attributes are also assigned their pre and post value, i.e., the XPath accelerator treats them like child nodes of the owning element node. Thus, a test on the node type will actually be compulsory for any evaluated location step.

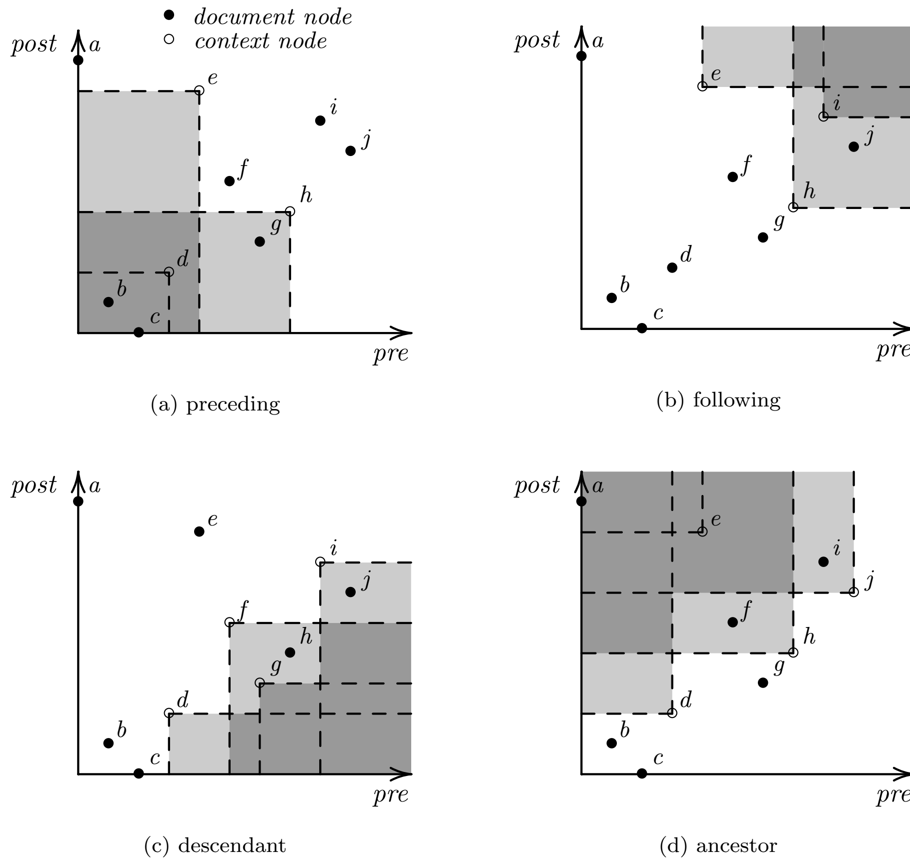


Figure 2.4: Overlapping regions (dark gray) in the pre/post plane for the XPath axes `preceding` (a), `following` (b), `descendant` (c), and `ancestor` (d).

2.3.1 Context Pruning

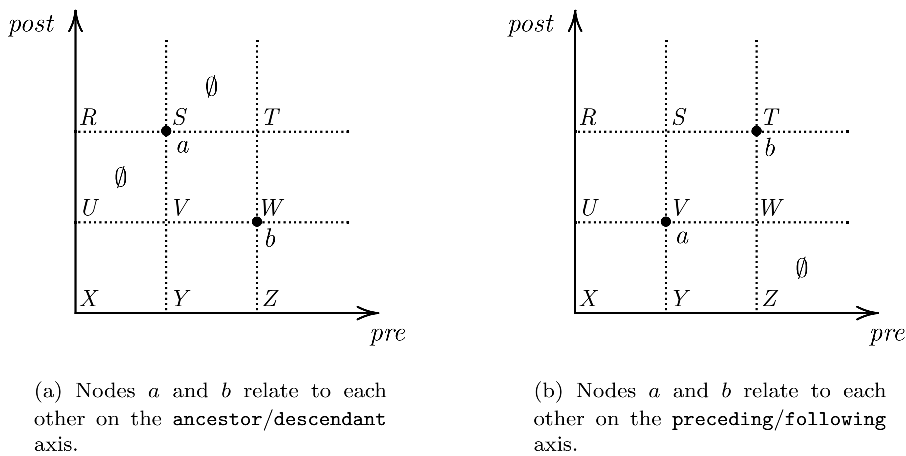
Context pruning is a technique which reduces the number of nodes in the context set to a minimum. This produces the following effects. First, the overall evaluation workload is decreased, since there are less context nodes to be considered during the evaluation of the region queries. Second, the overlap of the result regions in the pre/post plane is reduced to a minimum, which in turn reduces the number of duplicates in the obtained end result. In fact, for all axes except the `ancestor` axis, the result set produced by using a pruned context set is duplicate-free.

Context nodes are pruned on the basis of:

- *inclusion*, which means that the result region originating at context node v_1 is completely contained in the region originating at context node v_2, and

- *empty regions* in the pre/post plane, which are guaranteed not to contain any result nodes and which are caused by the relationship between the nodes in an XML document tree.

Empty Regions. Two nodes a and b divide the pre/post plane into nine regions denoted by the letters R to Z in Figure 2.5. Depending on the relationship between the two nodes, i.e. either `ancestor/descendant` or `preceding/following`, one or more of these regions are necessarily empty. If a is an ancestor of b such as in Figure 2.5 (a), regions S and U will always be empty. This is because any other ancestor c of b must either be an ancestor of a, too, or it must be a descendant of a. If we intersect the ancestor region of b with the ancestor and descendant regions of a, we find that only regions R and V are left to contain such nodes c. If the nodes a and b are related on the `preceding/following` axis such as in Figure 2.5 (b), region Z will always be empty, because it would have to contain common descendants of a and b which is impossible per definition, because b follows a.



(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 2.5: Empty regions in the pre/post plane.

Each XPath axis requires its own method of pruning. In case of the `preceding` axis, the context set can be reduced to one single node v_{max}, namely the context node with the *maximum pre value* (see node h in Figure 2.6). Any other context node v_i is redundant, because it falls into either one of the following two categories:

- v_i may itself be a preceding node of v_{max} (e.g. node d in Figure 2.6), which means that its result region is completely contained in the result region of v_{max}.
- Or v_i may be an ancestor of v_{max} (e.g. node e in Figure 2.6). If this is the case, empty region U makes sure that the preceding nodes of v_i are a subset of the preceding nodes of v_{max} (see Figure 2.6 (a)).

Similar to the `preceding` axis, the context set of the `following` axis can also be pruned to one single node v_{min}, namely the context node with the *minimum post value* (see node h in Figure 2.7). Any other node v_i is redundant, because it falls into either one of the following two categories:

- v_i may itself be a following node of v_{min} (e.g. node i in Figure 2.7), which means that its result region is completely contained in the result region of v_{min}.
- Or v_i may be an ancestor of v_{min} (e.g. node e in Figure 2.7). If this is the case, empty region S makes sure that the following nodes of v_i are a subset of the following nodes of v_{min} (see Figure 2.7 (a)).

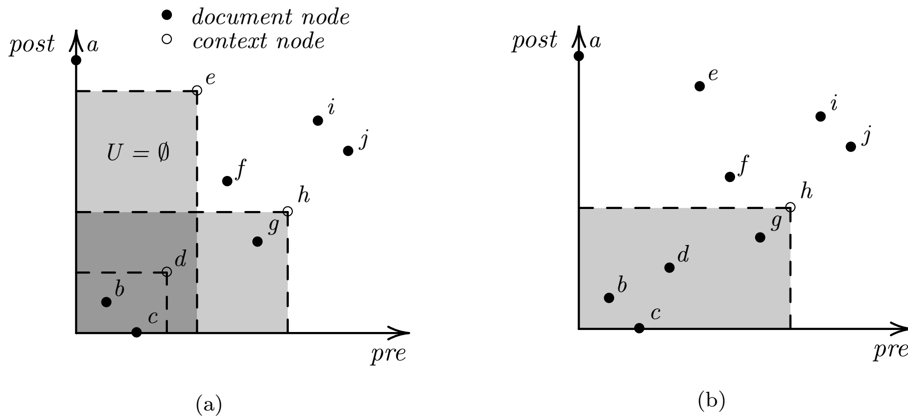


Figure 2.6: The pre/post plane before (a) and after (b) pruning for the `preceding` axis. The light gray area originating at node e corresponds to the empty region U as illustrated in Figure 2.5 (a) and thus cannot contain any result nodes.

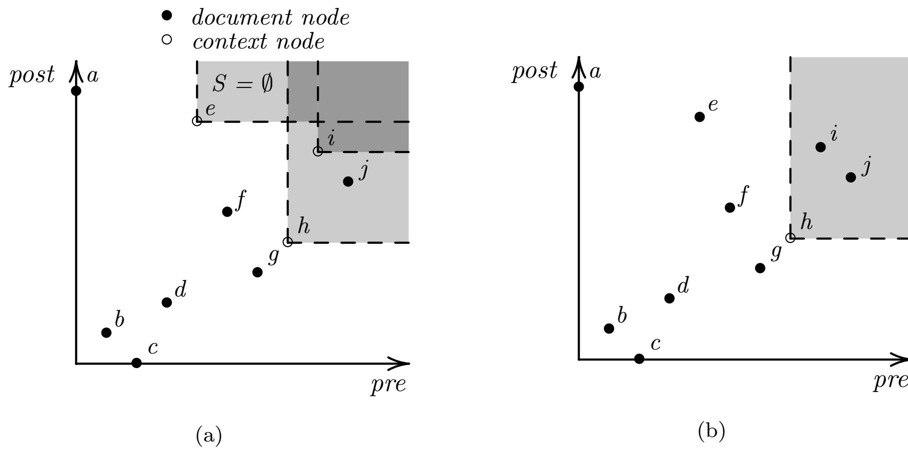


Figure 2.7: The pre/post plane before (a) and after (b) pruning for the `following` axis. The light gray area originating at node e corresponds to the empty region S as illustrated in Figure 2.5 (a) and thus cannot contain any result nodes.

In case of the `descendant` axis, pruning eliminates all context set nodes that are the descendant of any other context node in the set. Figure 2.8 illustrates that the descendant region of these nodes is completely contained in the descendant region of their ancestor (e.g. node g). The descendants of a node v have a higher pre and lower post value than v. The pruning algorithm depicted in Algorithm 1 encapsulates this knowledge. In a sequential scan of the pre-sorted context set, it removes all nodes that have a lower post value than their predecessors.

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

Algorithm 1: Context pruning for the `descendant` axis eliminates all context nodes that are the descendant of any other context node in the set. Table `context` is assumed to be *pre*-sorted.

The result of pruning for the `descendant` axis is illustrated in Figure 2.8 (b). The remaining overlap is necessarily empty, because it corresponds to the Z region shown in Figure 2.5 (b) (or a cluster of intersecting Z regions, if there are more than two context nodes left).

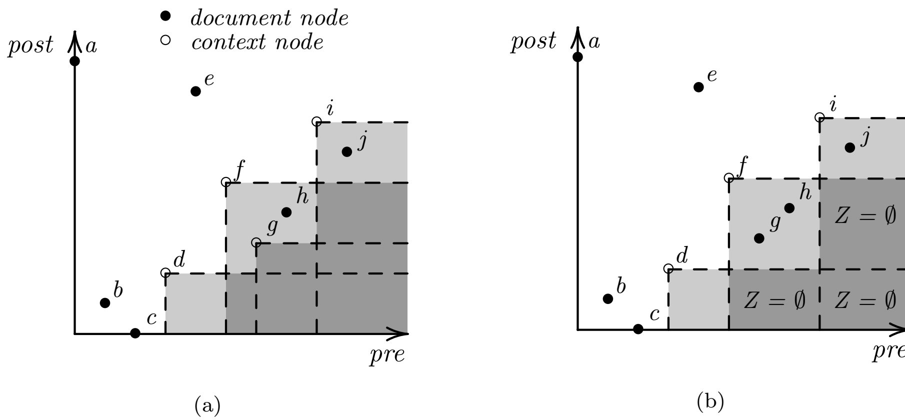


Figure 2.8: The pre/post plane before (a) and after (b) pruning for the `descendant` axis. The remaining overlap in (b) is empty, because it corresponds to a cluster of intersecting Z regions (see Figure 2.5 (b)).

Pruning the context set for the `ancestor` axis works similarly to `descendant` pruning. The algorithm eliminates all nodes which are the ancestor of any other context node in the set, because the ancestor regions of these nodes are completely contained in the ancestor regions of their descendants (e.g. node e in Figure 2.9). With respect to the remaining overlap, the `ancestor` axis represents a special case, because the overlapping regions may still contain nodes (e.g. nodes a and e in Figure 2.9 (b)). In this case, the mechanisms used in the staircase join itself will prevent the creation of duplicates.

Finally, note that pruning causes the boundary of the area spanned by the remaining context nodes to be shaped like a *staircase* (see Figures 2.8 and 2.9).

2.3.2 Partitioning and the Staircase Join Algorithms

Pruning reduces the number of context nodes and helps to avoid duplicates in the join result. In fact, as far as the `preceding` and `following` axes are concerned,

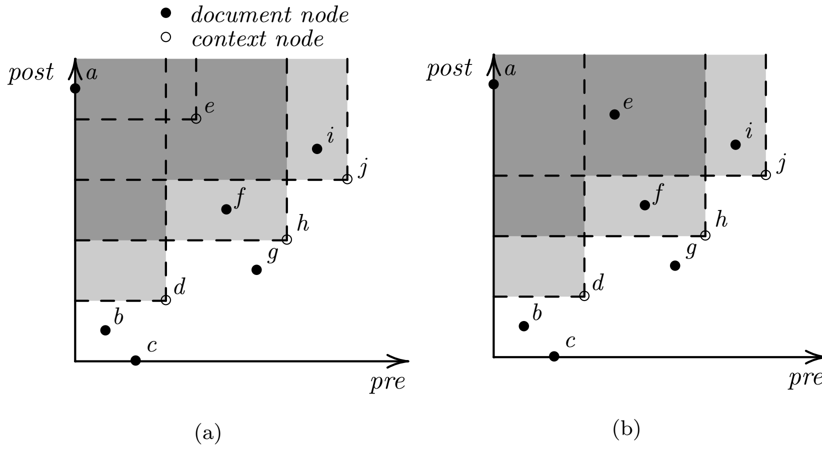


Figure 2.9: The pre/post plane before (a) and after (b) pruning for the `ancestor` axis. The remaining overlap may still contain nodes (e.g. a and e).

pruning has already led to the achievement of both optimization goals. Since there is only one context node left, pruning does not only prevent the creation of duplicates, but also reduces the number of document table scans to a minimum of one. The correct sort order of the result is ensured, if the document table is scanned in document order (i.e. in ascending *pre* order). Due to pruning, the evaluation of the region queries of the `preceding` and `following` axes could even be reduced to a pair of simple selections on the basis of the single context node. In any case, with pruning accomplished, these axes can be efficiently evaluated by a conventional join algorithm.

However, in case of the `descendant` and `ancestor` axes, a conventional RDBMS might nonetheless carry out a good deal of duplicate work. To be more precise, it is still likely to initiate several scans of the document table during the evaluation of the region queries. Apart from that, there remains the problem of duplicate result nodes in case of the `ancestor` axis. This is where the staircase join algorithms come into play. They make sure that one sequential scan of both input tables is enough to evaluate an XPath location step. To achieve this, the algorithms scan the document table in *partitions* of pre values, whose boundaries are defined by two successive nodes in the pruned context set. Figure 2.10 shows how partitioning works for the `ancestor` axis. Since each partition is scanned exactly once, this technique is also the key to avoiding duplicates in case of the `ancestor` axis.

The staircase join algorithms for the `descendant` and `ancestor` axes are depicted in Algorithm 2. For partitioning to work correctly, both of their input relations must be sorted on the pre value. In turn, the algorithms make sure that their output is sorted in the same way. The evaluated partitions lie between the pre values of context nodes v_1 and v_2. Exceptions are the last partition of the descendant algorithm, which ranges from the last node in the context set v to the very last node in the document table, and the first partition of the ancestor algorithm, which lies between the very first node in the document table and the first node in the context set v. The context set is assumed to have been pruned.

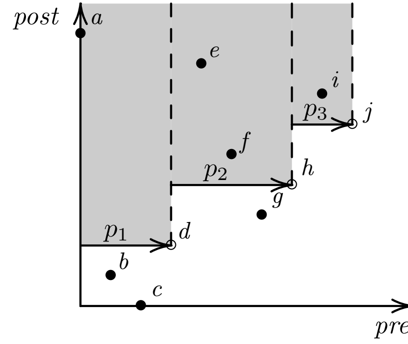


Figure 2.10: Partitioning in case of the `ancestor` axis. The document table is scanned partition-wise. The boundaries of each partition p_i are defined by two successive context nodes.

```
staircasejoin_desc (doc : TABLE (pre, post),
                            context : TABLE (pre, post)) ≡
BEGIN
  result ← NEW TABLE (pre, post);
  FOREACH SUCCESSIVE PAIR (v1, v2) IN context DO
    scan_partition (v1.pre + 1, v2.pre − 1, v1.post, >);
  v ← LAST NODE IN context;
  n ← LAST NODE IN doc;
  scan_partition (v.pre + 1, n.pre, v.post, >);
  RETURN result;
END

staircasejoin_anc (doc : TABLE (pre, post),
                           context : TABLE (pre, post)) ≡
BEGIN
  result ← NEW TABLE (pre, post);
  v ← FIRST NODE IN context;
  n ← FIRST NODE IN doc;
  scan_partition (n.pre, v.pre − 1, v.post, <);
  FOREACH SUCCESSIVE PAIR (v1, v2) IN context DO
    scan_partition (v1.pre + 1, v2.pre − 1, v2.post, <);
  RETURN result;
END
```

Algorithm 2: The staircase join algorithms for the `descendant` and `ancestor` axes.

The evaluation of the region queries takes place within the `scan_partition()` routine as illustrated in Algorithm 3. The first clause of the region queries (the *pre clause*) is implicitly covered by the *for loop*, which makes sure that we stay within the partition boundaries. The second clause (the *post clause*) is evaluated in the *if* branch within the loop. Note that the actual context node on which the computation of the result nodes is based is represented by v_1 in case of the

`descendant` axis and v_2 in case of the `ancestor` axis.

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

Algorithm 3: Algorithm responsible for evaluating the pre and post clause within a specified partition. Table `doc` represents the current instance of the document table and table `result` was initialized by the calling algorithm (see Algorithm 2).

2.3.3 Skipping

Skipping is an additional technique which minimizes the number of nodes in the document table that must be accessed during the scan of that relation. It is based on the empty region observations made in Figure 2.5 and applies to the `descendant` and `ancestor` axes. Remember that, in both cases, the document table is scanned partition-wise in ascending *pre* order, i.e. from left to right in the pre/post plane.

Skipping for the Descendant Axis

Suppose that the partition between context nodes v_1 and v_2 is currently scanned. As soon as we come across the first node n with a higher post value than v_1 within the partition, we have found the first following node of v_1 and may skip the remaining document nodes in the current partition. This is because v_1 and n relate to each other on the `preceding/following` axis and their Z region, which is actually part of the staircase, is guaranteed to be empty (see Figure 2.11).

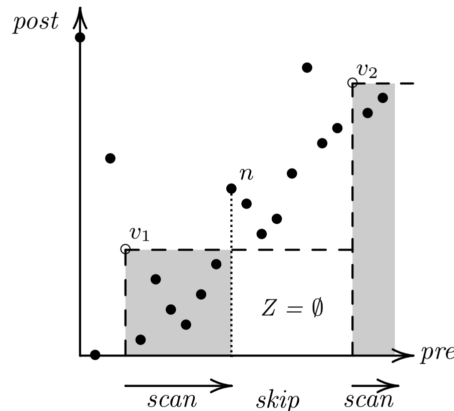


Figure 2.11: Skipping technique for the `descendant` axis. As soon as we come across the first following node of v_1, we may skip the remaining document nodes in the current partition.

In case of the `descendant` axis, skipping can be incorporated into the algorithm `scan_partition()`. As soon as the first node with a higher post value is found, the

evaluation of the current partition is terminated and the algorithm proceeds with the next partition:

```
scanpartition (pre_1, pre_2, post, >) ≡
BEGIN
  FOR i FROM pre_1 TO pre_2 DO
    IF post > doc[i].post THEN
      APPEND doc[i] TO result;
    ELSE
      BREAK;    /* skipping */
END
```

Algorithm 4: Skipping algorithm for the `descendant` axis.

For the `descendant` axis, skipping effectively limits the number of accessed tuples to at most $|result\ nodes| + |context\ nodes|$. In other words, it makes sure that at most one tuple is accessed per partition (i.e. context node) which is not part of the result.

Skipping for the Ancestor Axis

For the `ancestor` axis, a slightly different approach is used. If we come across a preceding node n of context node v during the evaluation, we know that all descendants of n are preceding nodes of v as well and may thus be skipped. The number of descendants of a node n can be obtained with the following equation:

$$|(n)/\texttt{descendant}| = post(n) - pre(n) + level(n).$$

However, as we do not have a means to obtain the tree level of n ($level(n)$) during staircase join execution, the number of skipped nodes for the `ancestor` axis is estimated to amount to $n.post - n.pre$. Since $level(n)$ is restricted by the overall height $h(t)$ of the XML document tree, the maximum amount by which the estimation may go wrong is $h(t)$. This is negligible, if we compare the height of existing XML document trees to the number of nodes contained within them. In our test environment, for example, $h(t)$ constantly amounts to 11, while the number of nodes ranges between 5,000 and 50 million.

Chapter 3

Implementation of the Staircase Join in PostgreSQL

This chapter provides an insight into the implementation of the staircase join in PostgreSQL. It starts off with an overview of the query processing stages in the PostgreSQL backend and proceeds with a few table- and query-specific restrictions that facilitate and support the upcoming implementation tasks. The following three sections of the chapter are dedicated to the implementation steps that were undertaken in the parser, planner/optimizer, and executor module of PostgreSQL, respectively.

More detailed information on the most important data structures used in these modules and a selection of routines, that were added or modified in the course of the implementation, can be found in Appendices A to D. The complete source code of the tree-aware PostgreSQL database can be found on the CD included in this thesis. The majority of explanations and illustrations in the following sections are based on the example XPath query presented in Section 2.2.2.

3.1 Query Processing in the PostgreSQL Database Engine

PostgreSQL is an object-relational, open-source database management system which offers support for the `SQL92/SQL99` standard and other features such as inheritance, user-defined data types, and user-defined functions.

The foundations for PostgreSQL were laid in 1986 in the course of the POSTGRES research project at the Berkeley Computer Science Department of the University of California. In 1994, the POSTGRES-specific query language PostQUEL was replaced by SQL which ultimately resulted in the new name of PostgreSQL. Today, the development of the DBMS is driven by the PostgreSQL Global Development Group, a community of companies and individual people. (For more details refer to the PostgreSQL web site at `http://www.postgresql.org`.)

The following sections describe the four successive stages through which a query must pass while being processed by the PostgreSQL database engine. For reasons of simplicity, the sections focus on SQL `SELECT` statements which consist of selections, projections, and joins only.

3.1.1 The Parser

The PostgreSQL parser verifies whether an SQL statement is syntactically correct. If this is the case, an internal representation of the statement, the *parse tree*, is created and passed on to the next stage. In PostgreSQL, parsing involves three steps:

– On the basis of regular expressions, the *lexical analyzer* (*lexer*) filters SQL keywords and identifiers from the query string and converts them into tokens.

– The *parser* uses the tokens produced by the lexer to validate the query against the grammar of the SQL syntax and builds up the parse tree.

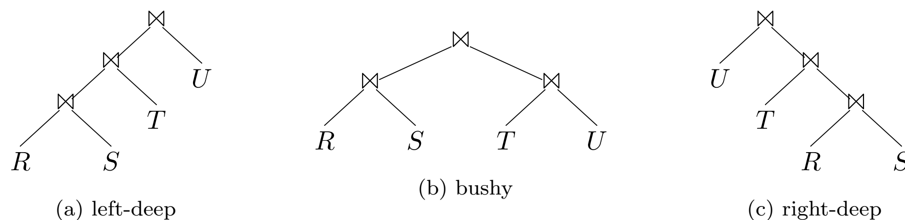
– The *transformation process* converts the parse tree into a normalized internal form, the `Query` structure (see Appendix A.1), which is independent of the type of the original SQL statement (`SELECT`, `INSERT`, `DELETE`, etc.) and in which the names of relations, attributes, and operators have been converted into numeric identifiers. During this process, PostgreSQL also makes sure that all names are known to the system and used "in scope".

3.1.2 The Rewriter

PostgreSQL allows the user to define rules which are to be executed in case of a specified event (`CREATE RULE`). If such a rule affects the currently processed query, its parse tree must be rewritten according to the action part of the rule. Rules are particularly useful for the implementation of views in PostgreSQL. (For more details see [Sim98].)

3.1.3 The Planner/Optimizer

The PostgreSQL planner/optimizer is concerned with the search for a query's optimal execution plan. In case of *single-relation queries*, the optimal plan corresponds to the cheapest access path of the relation. The access methods available in PostgreSQL are *sequential scan* and *index scan* (provided a suitable index on the relation exists). In case of *multi-relation queries*, joins come into play which results in two additional cost factors: the order in which the relations are joined (join tree) and the algorithm used to execute the join. PostgreSQL supports *left-deep*, *right-deep*, and *bushy join trees* as illustrated in Figure 3.1. The currently supported join algorithms are *nested-loop join*, *merge join*, and *hash join*.



(a) left-deep (b) bushy (c) right-deep

Figure 3.1: Three types of join trees for the relational-algebra query $R \bowtie S \bowtie T \bowtie U$.

Join Algorithms. In a *nested-loop* join, the inner relation is scanned once per tuple in the outer relation.[a] Consequently, this join algorithm does not require any sort order of its input relations. However, if a suitable index on the inner relation is available, PostgreSQL uses the current outer tuple as variable index search key to initiate a so-called *index rescan* which makes sure that only matching inner tuples are retrieved. In contrast to the nested-loop join, the *merge join* requires both of its input relations to be sorted on the join attributes (or a subset hereof). In such a join, both relations are scanned only once. In PostgreSQL, the merge join only supports *equi-joined* clauses, i.e., clauses in which the operands are connected by the equals sign ("="). In a *hash join*, the inner relation is scanned once to sort its tuples into a hash table. After that, the hash function is applied to the tuples in the outer relation to retrieve the matching inner tuples from the resulting hash bucket. Due to the nature of hashing, this join algorithm is also restricted to equi-joined clauses.

[a] In the join tree, the *outer* relation corresponds to the left child and the *inner* relation to the right child of a join node.

The execution plan for multi-relation queries is determined in the course of a dynamic programming algorithm. It considers all combinations of join orders and join methods while incrementally joining the base relations. The first pass of the algorithm is based on the cheapest access paths of the joined relations. Similarly, each subsequent pass takes only sub-optimal results from previous passes into account. The cheapest overall result is converted into a plan and passed on to the executor. Figure 3.2 shows one example plan that might be created for the query in Figure 3.1.

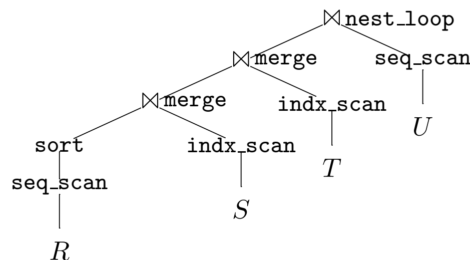


Figure 3.2: Example of an execution plan for the query in Figure 3.1.

3.1.4 The Executor

The plan created by the planner/optimizer instructs the executor by which method and in which order the query's base relations must be accessed and joined (see Figure 3.2). The executor takes three separate passes at the plan tree. Each of them starts at the top plan node. The first pass is dedicated to various initializations, e.g. the initialization of internal execution states and the allocation of memory for storing the input tuple(s) and the result tuple. Each parent node in the plan recursively triggers the initialization of its child nodes. In the second pass, the actual execution of the plan takes place. It is entirely focused on *pipelining* (though materialization is, of course, also possible). This means that a parent node only requests the next input tuple from a subplan, if the tuple is immediately required to continue processing. If the subplan has no appropriate tuple ready, it must in turn trigger the execution of one of its subplans and so forth. This process may continue until a

leaf node is reached and the next tuple is fetched directly from a base relation. In this way, parts of the plan may be recursively traversed various times. Finally, the third pass is dedicated to a general clean-up, e.g. the deallocation of memory etc.

3.2 Restrictions in a Tree-Aware PostgreSQL Instance

The implementation described in this thesis claims by no means to be complete. First and foremost, it is beyond the scope of this thesis to seamlessly integrate the staircase join with all kinds of SQL constructs. This is why we allow only a reduced syntax for SQL-based XPath queries. Besides selections, projections, and joins, they may contain claims for distinctness and sort order to make the result tuples comply with the XPath specification. Other SQL constructs, such as nested queries, SQL `JOIN` expressions, etc. were not considered.

In order to decide whether an SQL query represents a translated XPath expression — and thus whether the staircase join qualifies to answer it —, we will examine or presuppose a number of table- and query-specific characteristics which are described in the following sections.

3.2.1 Table-Specific Restrictions

The relations involved in an SQL-based XPath expression must fulfill the following requirements to ensure that the staircase join is planned and executed properly.

- The context set and document tables must contain at least two columns, the pre and the post column, which must be named accordingly ("pre" and "post"). The pre column must be the first column of either table and the post column the second. Both attributes must be assigned the special data type `tree`, which was newly introduced during the implementation (see Section 3.3).
- The context set and the document table must have the same schema, including the data types of the columns. This is due to an inflexibility in the way in which PostgreSQL evaluates join clauses. It reserves one tuple slot for working with tuples from the outer relation and one slot for tuples from the inner relation. However, the staircase join sometimes requires two outer or two inner tuples to be compared. As a work-around, both slots must have been assigned the same tuple descriptor (i.e. schema).
- Since we would like the evaluation of an SQL-based XPath expression to start at the context set table, PostgreSQL must be enabled to recognize it. Thus, it is demanded to carry the substring "context" somewhere within its name.

3.2.2 Query-Specific Restrictions

The translated XPath expressions sent to the tree-aware PostgreSQL instance are identical to the queries described in Section 2.2.2. In order to determine whether a query represents such an expression, a close examination of the join clauses will be carried out during query processing. The examination will not only tell us whether the staircase join can be used to answer the query, but also deliver the XPath axis represented by the clauses.

The following restrictions apply to the join clauses. First, the context set — either the explicitly specified, initial context set table or the document table instance which links the current join to the previous one — must always be referenced within

the left operand of a join clause. Applied to our example query, this means that in the first join *cs* and in the second join doc_1 must be the left operand. This facilitates the identification of the represented location step. Second, there may be no additional join clauses that involve comparisons of pre or post values. Further XPath-specific predicates, such as name tests etc., take the form of selection clauses.

3.2.3 Availability of Indices

A further important requirement is the existence of an index on the document table. As we will see, it is the key to an efficient implementation of partitioning and skipping in PostgreSQL, because it allows us to jump directly to a required tuple. The index must have the pre value as primary index key and may include further attributes on demand. Note, however, that the post value must NOT be part of the index. This is because we must be able to distinguish between the pre and the post clause during staircase join execution. If both clauses were evaluated within the index, there would be no means to do so and essential components of the staircase join (skipping and partitioning) could not be applied.

3.3 Implementation Steps in the Parser

Most of the work involved in the implementation of the staircase join in PostgreSQL takes place in the planner/optimizer and the executor. However, there is also a modification concerning the parser. It is supplied with an additional data type named `tree`. This involved the introduction of a new keyword into the lexer and the corresponding terminal symbol (`TREE`) into the SQL grammar of PostgreSQL. Apart from that, the characteristics of the new data type had to be added to the system catalog, including a number of procedures which specify how to read in and print out data of type `tree` and how to evaluate operator expressions for it.

The new data type is reserved for table columns that contain pre or post values. It is intended as a means to indicate that a relation contains tree-specific data. Since pre and post values are actually integer values, the new data type was modeled as a derivative of the data type `int`, i.e., it inherits all its associated properties and procedures from this type.

3.4 Implementation Steps in the Planner/Optimizer

The aim of the implementation work in the planner/optimizer was to make PostgreSQL consider the staircase join as join algorithm during dynamic programming and to integrate it into the final execution plan, if appropriate. In order to determine whether the staircase join can be used for the execution of a join, the join clauses must be closely examined. To avoid redundant sorting steps, we must also keep track of the sort orders associated with the input relations and the result of the join.

A detailed description of the most important data structures involved in planning and optimization can be found in Appendix A. Appendix B gives an overview of the routines that are involved in creating a staircase join execution plan. It also provides references to the locations in which these routines can be found on the CD and/or the online appendix which complement this thesis.

3.4.1 Clause Preparation

In case of a basic query, such as an SQL-based XPath expression, one of the first tasks in the planner/optimizer is dedicated to the preparation of the clauses that

occur in the query. During this process, each of the predicates in the `WHERE` part is examined and classified as selection or join clause.

Selection clauses ($R.X = 5$) and clauses that compare two attributes from the same relation ($R.X = R.Y$) are distributed to the base relation to which they refer (R). The PostgreSQL executor will evaluate them during the scan of that relation (*selection pushdown*). They are not of interest for the staircase join implementation.

Join clauses require a more thorough examination. Depending on the data types of their operands and the connecting operator, PostgreSQL determines in which type of join a clause can be used. For example, it was indicated before that merge and hash joins are restricted to equi-joined clauses. To identify a potential *staircase join clause*, our implementation makes sure that its operator is one of the *staircase join operators* (“$<$” or “$>$”) and that both operands are of the newly introduced `tree` type. Figure 3.3 shows the internal representation of the two pre clauses of our example query after this process. The “join type” field specifies in which type of join they can be used.

clause		
join type:	`strcs`	
operator:	$<$	
left operand:	relation:	*cs*
	attribute:	*pre*
	type:	`tree`
right operand:	relation:	doc_1
	attribute:	*pre*
	type:	`tree`
left sort key:	($cs.pre$, asc)	
right sort key:	($doc_1.pre$, asc)	

(a)

clause		
join type:	`strcs`	
operator:	$<$	
left operand:	relation:	doc_1
	attribute:	*pre*
	type:	`tree`
right operand:	relation:	doc_2
	attribute:	*pre*
	type:	`tree`
left sort key:	($doc_1.pre$, asc)	
right sort key:	($doc_2.pre$, asc)	

(b)

Figure 3.3: Internal representation of two pre clauses $cs.pre < doc_1.pre$ (a) and $doc_1.pre < doc_2.pre$ (b).

Note that this process does not yet predefine which type of join will be chosen in the end. It just determines which clause qualifies for which join variant. A nested-loop join can be used for any clause.

If a join method requires its operands to have a particular sort order, this information will also be recorded here. For example, a staircase join clause specifies that both of its operands must be sorted on ascending pre values.

The routines involved in clause preparation are listed in Appendix B.1.

3.4.2 Preparation of Access Paths

For each relation that occurs in a query, PostgreSQL must identify the available access paths. For reasons of efficiency, it was demanded in Section 3.2 that there is an index on, at least, the pre value of the document table. PostgreSQL knows two types of index scans:

- “Normal” index scans where a constant value is used to access the first index entry in the scan. All subsequent index entries are retrieved sequentially. This type of index is used for selections, the merge join, and sorting.

– *Inner-join index* scans where several rescans of the index can be initiated and the search key for each index rescan is variable.

  Assume that there is the join clause $R.X < S.Y$, R is the outer and S the inner relation and S has an index on attribute Y. In an inner-join index, every new attribute value of $R.X$, e.g. 17, can be used as index search key to trigger a rescan of S which is guaranteed to start directly at the first tuple with $S.Y > 17$. For these indices to work properly, the join clause $(R.X < S.Y)$ is converted into a special kind of index clause which is incorporated into to the inner-join index. When an index rescan is initialized, the executor looks up the current value of $R.X$ and retrieves all matching entries from the index.

Inner-join index scans have so far only been used in connection with the inner relation of a nested-loop join (index nested-loop). They make sure that the number of scanned inner tuples is reduced to a minimum. However, the inner-join index also turned out to be important for the implementation of the staircase join. The idea is to incorporate the evaluation of the pre clause into the index. In this way, we can use the pre value of the current context node to jump directly to the first document node matching the pre clause. As we will see, this possibility is essential for an efficient implementation of partitioning and skipping. Figure 3.4 shows an example of an inner-join index for relation doc_1.

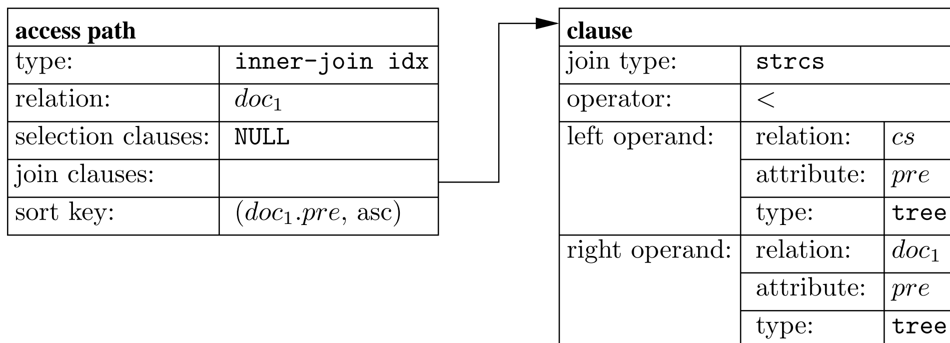


Figure 3.4: Access path representing an inner-join index on the *pre* value of relation doc_1.

To adapt this kind of index for the use in a staircase join, there was only a slight modification necessary. Once an index has been identified as candidate for an inner-join index, its sort order no longer matters to a nested-loop join. Thus, the sort keys of the index were previously not recorded. Since the staircase join relies on this piece of information, it was additionally included into the inner-join index (see Figure 3.4).

3.4.3 Marking the Context Set

In the query-specific requirements of an SQL-based XPath query in Section 3.2, we determined that the table representing the initial context set must carry the substring "context" in its name. We make use of this property now and mark the relation accordingly. During dynamic programming, it will be the starting point for the creation of the query's join tree.

3.4.4 Dynamic Programming

During its dynamic programming algorithm, PostgreSQL incrementally builds up the join tree of the processed query. The first pass of the algorithm takes into account all possible ways to join two base relations, while each subsequent pass either joins a base relation and the result of a previous join (linear join tree) or two join results (bushy join tree).[3] Suppose the algorithm is currently considering a join between relations R and S. After having retrieved the clauses that apply to the join, the database considers each of the three available join algorithms. This process is carried out twice, once regarding R as outer and S as inner relation and once vice versa. If one of the join variants is found to be applicable, a new join tree node is created and the associated cost is estimated. The cheapest one of the newly created nodes is stored and used as input parameter in the subsequent pass. Eventually, the cheapest overall join tree is selected, converted into a plan and passed on to the executor.

The staircase join implementation must make sure that PostgreSQL also considers the creation of staircase join nodes during dynamic programming. The following paragraphs explain the implementation steps involved in this process. The result of dynamic programming for the join between cs and doc_1 is illustrated in Figure 3.5. The routines involved in the creation of a staircase join node can be found in Appendix B.2.

The staircase join differs from the three native PostgreSQL join algorithms in two important respects. First, it imposes a predefined join order which corresponds to the order of location steps in the original XPath expression.[4] Applied to our example query, this means that the join between cs and doc_1 (the `following` axis) must be executed before the join between doc_1 and doc_2 (the `descendant` axis). Second, it makes no difference to the cost of a staircase join which of the input relations (e.g. cs or doc_1) is considered the outer and which the inner relation. This is because, internally, the staircase join algorithm will always treat cs as the current context set and doc_1 as an instance of the document table. Thus, it suffices to build *either* left-deep *or* right-deep trees. Bushy trees are not taken into account at all.

In the implementation, we arbitrarily decided in favor of left-deep trees. Consequently, any join tree created for a series of staircase joins will have the structure illustrated in Figure 3.6. The starting point of such a join tree is the explicitly specified context set table. It takes the place of the outer relation. In any subsequent pass of dynamic programming, the result of the previous join becomes the outer relation. The inner relation is always represented by a base relation, i.e. a new instance of the document table. The join order is predefined by the join clauses. Starting at the initial context set guarantees that it corresponds to the original order of location steps.

Assuming that dynamic programming is currently considering the creation of a new staircase join node (e.g. for the join between cs and doc_1), PostgreSQL must first identify the pre and the post clause and use them to determine the represented XPath axis. First, the names of the two attributes involved in each clause are examined. If both are named “pre”, the predicate is a candidate for the pre clause. If both are called “post”, the predicate is likely to be the post clause. If two candidate clauses have been successfully identified, it is made sure that they have been marked as being usable in a staircase join. If this is the case, the operator combination of the two join clauses is used to determine the given XPath axis.

[3] In the following, we will simply refer to both input parameters of a new join node as *(input) relations*. More precisely, the left input parameter will be called *outer* and the right parameter *inner relation*.

[4] Actually, the optimal join order should be determined on the basis of a cost model designed for the staircase join. However, this would have been beyond the scope of this thesis, such that the implementation only allows the join order predefined by the location steps.

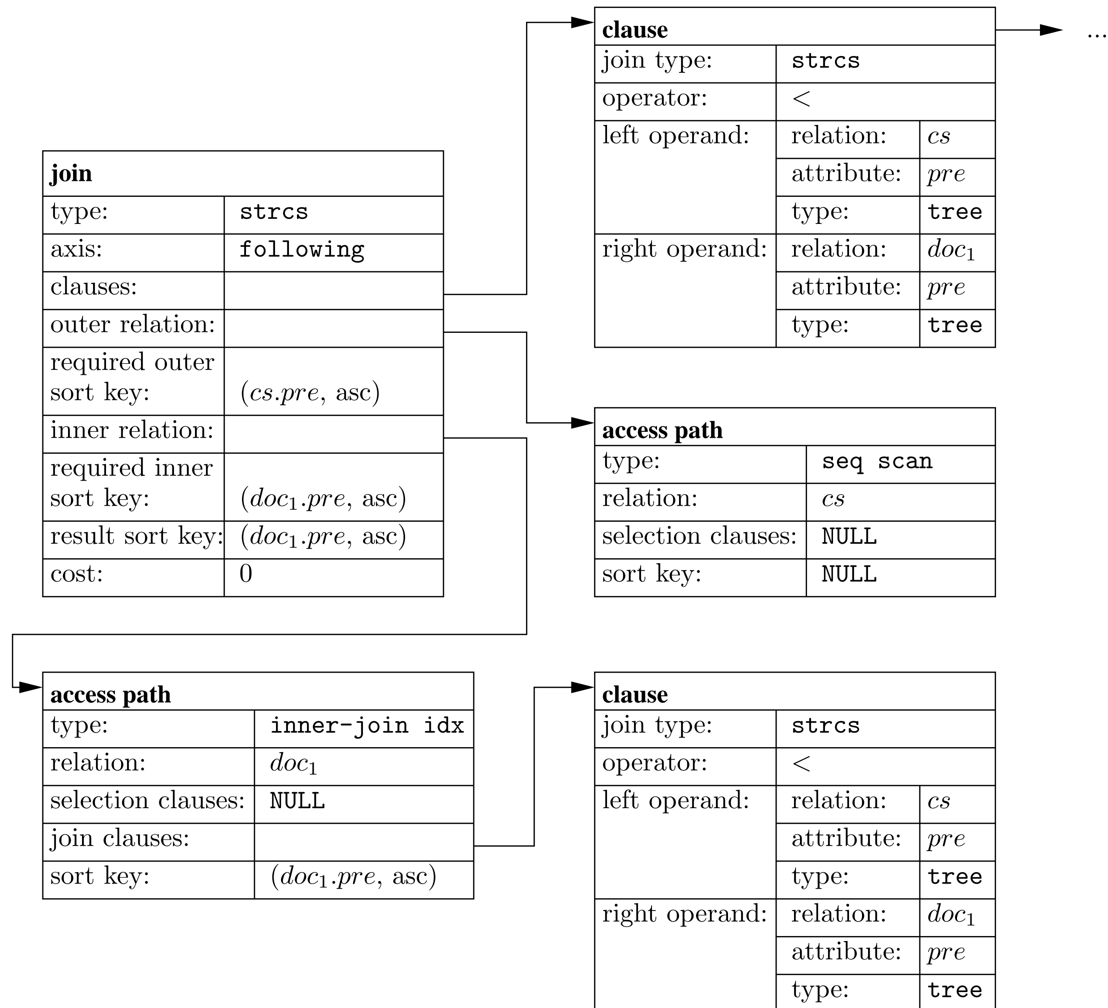


Figure 3.5: The result of dynamic programming for the join between *cs* and doc_1. The represented XPath axis was identified and the required sort keys were recorded.

For example, a combination of two lower than operators (“<”) indicates that the currently considered join represents the `following` axis. The correctness of the result is ensured by the query-specific requirement that the left operand of a clause must refer to the current context set.

Keeping Track of Sort Orders

When a new staircase join node is created, it is also important to keep track of sort orders. The sort order required of its two input relations is recorded in the representation of the pre clause. For relations *cs* and doc_1, we will, for example, obtain a sort order of (*cs*.*pre*, `asc`) and (doc_1.*pre*, `asc`), respectively (see Figure 3.3 (a)).

If one of the relations is found *not* to have the required sorting, an explicit sort node will be inserted during the creation of the final execution plan. Note that an additional sorting step for the inner relation (the document table) will never be required. Since it must be available as an inner-join index with the pre value as

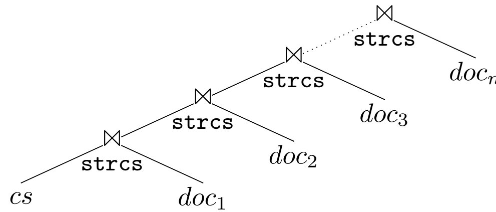


Figure 3.6: Typical structure of a join tree created for a series of staircase joins. The right input relation of the join is a new instance of the document table, while the left input parameter stands for the context set — either the initial context set *cs* or the result of the previous join.

primary key, its available sort key will always correspond to the required one.

The result of the staircase join is in turn guaranteed to be sorted on the pre value of the document table doc_1. Thus, we can inherit the sort order of doc_1 to the join result. As this is exactly the sorting required of the outer relation in the subsequent join between doc_1 and doc_2 (see Figure 3.3 (b)), the inheritance of sort keys additionally makes sure that PostgreSQL always finds it to be suitably sorted. An exception is, of course, the initial context set in the very first staircase join. It is the only case in which an explicit sorting step might have to be inserted.

Estimating the Execution Cost

Finally, the execution cost must be estimated for the staircase join. It has already been indicated that writing a cost function for a new join algorithm is beyond the scope of this thesis. Therefore, a cost factor of 0 is used for such joins. This makes sure that the staircase join is always the algorithm of choice in later performance tests.

3.4.5 Conversion into an Execution Plan

Once the cheapest join tree has been determined, it is recursively traversed and converted into an execution plan. Starting at the leaf nodes, the process transforms each join tree node into a plan node, a so-called *subplan*. There is a one-to-one correspondence between the two node types. Index scan nodes are converted into index scan subplans, staircase join nodes into staircase join subplans, etc. However, before a staircase join subplan can be created, the join clauses must be prepared for the clause evaluation mechanisms of the PostgreSQL executor. Apart from that, explicit sort nodes must be inserted into the plan, if necessary. Figure 3.7 shows the execution plan created for the join between *cs* and doc_1. The routines involved in the conversion into a staircase join execution plan can be found in Appendix B.3.

Preparation for Clause Evaluation

When a staircase join subplan is created, the planner/optimizer has made sure that an inner-join index is used to access the inner relation. Consequently, a copy of the pre clause has been additionally stored as index clause (cf. Figure 3.4). To prevent the clause from being evaluated twice — once by the index and once by the join — it must now be removed from the list of clauses in the join plan.

To prepare a clause for the evaluation mechanisms of the PostgreSQL executor, the references to the relations that occur in the clause must be changed. When a

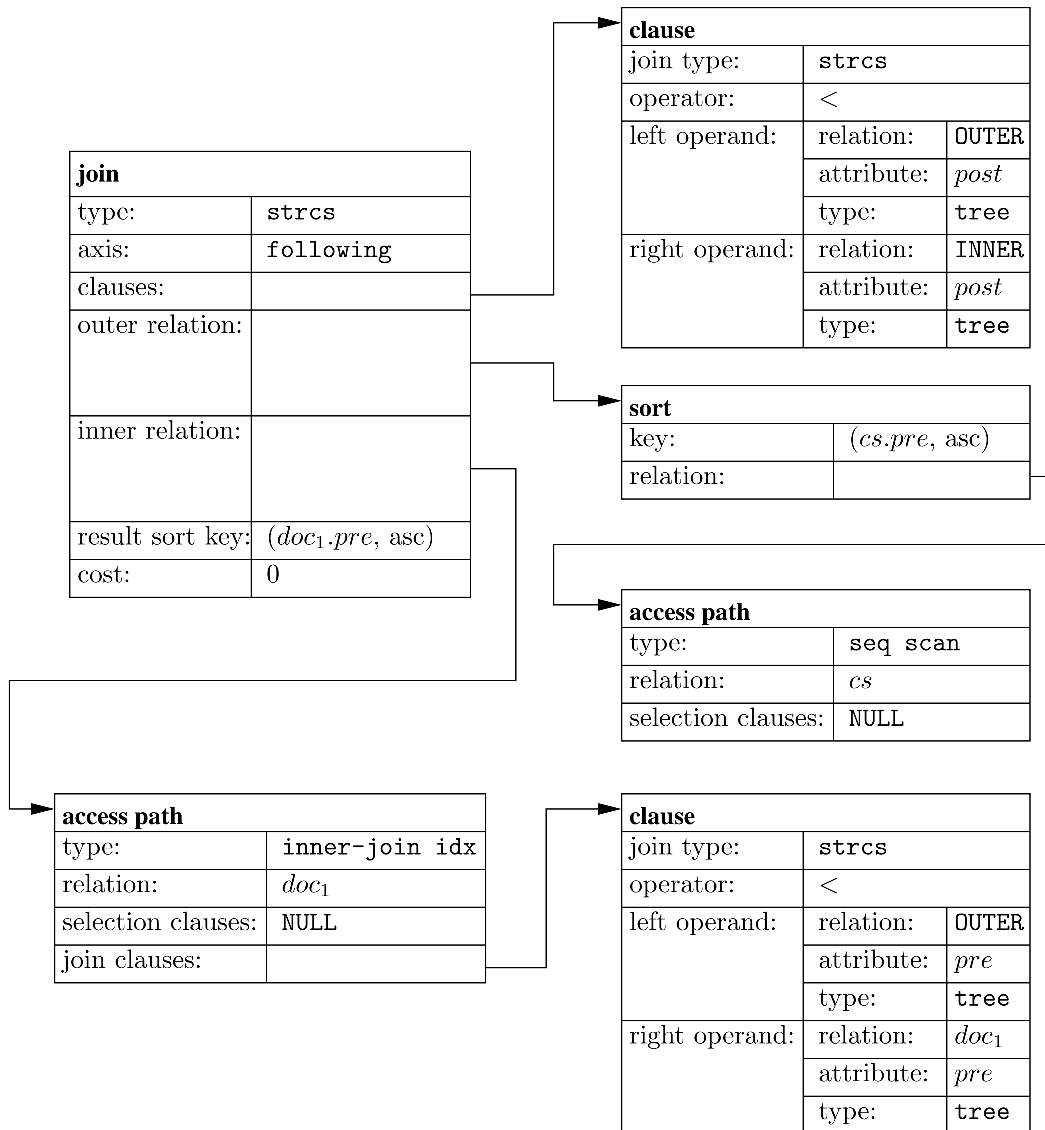


Figure 3.7: The execution plan created for the join between *cs* and doc_1. The pre clause was removed from the list of join clauses and the references to the involved relations were changed.

clause is evaluated, the executor does not require the exact identification (e.g. *cs* or doc_1) of a relation, but just needs to know whether the operand refers to the `OUTER` or `INNER` relation (see Figure 3.7). This is because the evaluation mechanisms know nothing about relations, but exclusively work on tuples which originate from the *outer* and *inner* subplan. Figure 3.7 shows that this approach is also applied to the variable search key operand of an index clause. Here, PostgreSQL only needs the outer tuple for the evaluation of the clause. The inner tuple is not yet known at this point. It is the aim of the evaluation to retrieve it.

Insertion of Sort Nodes

Finally, if one of the input relations is not suitably sorted, a sort node is inserted into the execution plan. In case of an SQL-based XPath query, this step may only be required for the initial context set table which is generally accessed in a sequential scan.

3.4.6 Distinctness and Sort Order of the Final Result

After the join tree and the associated execution plan have been created, the explicit claims for sort order (`ORDER BY`) and distinctness (`DISTINCT`) of the result tuples must be processed. If necessary, a sort node and, on top of that, a so-called *unique* node are inserted into the plan. In case of a staircase join plan, the insertion of these additional nodes can be tied to certain conditions.

We know that the result of a staircase join is sorted on the pre value. If this sort key turns out to be a prefix of the sort keys required by an `ORDER BY` clause, there is no need for an explicit sort node on top of the staircase join node. This is because the join result is also known to be duplicate-free which makes any secondary sort key redundant. The distinctness of the staircase join's result may also abolish the need for a unique node. It can be left out, if the top plan node is a staircase join node, i.e., if the result tuples may be returned in the "natural" sort order in which they are produced by the staircase join.

3.5 Implementation Steps in the Executor

The plan is now ready to be executed. Since each type of subplan has its own execution module, a totally new module responsible for the execution of the staircase join was introduced. It embraces implementations of pruning, partitioning, and skipping. Since the original staircase join algorithms specify the materialization of the join result, one important task was to adapt them to the pipelining mechanisms of PostgreSQL. Another important aspect was to ensure the optimal use of the available inner-join index.

Before we proceed, a word on the `preceding` and `following` axis remains to be said. In Section 2.3.2, we have announced that the evaluation of these two axes could be reduced to a selection, because there is only a single context node left after pruning. However, this direction could not be followed through in PostgreSQL. For once, the database has automatically classified the pre and the post clause as join clauses, because they reference two distinct relations and both of their operands are variable. A solution might be to rewrite the join clauses. But in the executor, it is too late to do so, because we would not only have to rewrite the clauses, but also reassign them to the respective access paths (selection pushdown), and in the planner/optimizer, it would have been too early, because we did not know yet what the pre and post value of the context tuple would eventually be.

However, executing the evaluation of the `preceding` and `following` axes as a join, still leaves two alternatives for the implementation. We could either separate

pruning and join execution, which would allow us to apply pruning first and then use PostgreSQL's native join algorithm (the nested-loop join) to evaluate the location step. Or we could incorporate both, pruning and join execution, into the staircase join execution module.

The previous sections have already indicated that the second alternative was followed through in the implementation. One of the main reasons was the desire for a uniform representation (the staircase join) for the evaluation of an XPath location step in PostgreSQL. Consequently, the `preceding` and `following` axes were implemented very similarly to the join algorithms of the `descendant` and `ancestor` axes. As we will see in Section 3.5.4, the main difference is that they work on one partition only.

3.5.1 Initialization of Staircase Join Execution

Before a plan can be executed, all of its subplans must be initialized in a recursive descent of the plan tree, i.e., each parent plan triggers the initialization of its subplans. Since the essential data structure in the executor is the *tuple*, the most important initialization task is the reservation of table slots in which the processed tuples can be stored. A join node is typically assigned one slot for the current outer, one for the current inner tuple[5] and another one for the result tuple. Apart from that, the staircase join requires a further table slot to store a previously processed tuple. For example, it will be needed to hold the second boundary of a partition. The routine responsible for the initialization of the staircase join can be found in Appendix D.1.

3.5.2 Pipelining

The pipelining policy of PostgreSQL demands that each node returns exactly one tuple to the caller (i.e. the parent plan node) and only triggers the execution of its subplans when a tuple from that plan is immediately required for execution. If one of the leaf nodes has run out of tuples, this status is propagated upwards until it reaches the top plan node. It indicates that execution is complete.

Suppose the database is processing a single join between relations R and S, the join clause is $R.X = S.Y$. After the join node has requested the first tuple from both, R and S, it proceeds with the evaluation of the join clause. If it is satisfied, the join is executed and, after the appropriate projections have been applied, the result tuple is returned. Depending on the selected join algorithm, the join node may then resume execution in different manners. In case of a nested-loop join, the next inner tuple is requested from S, while a merge join would typically request a new tuple from both input relations.

The original staircase join algorithms specify the materialization of the join result. However, PostgreSQL avoids materialization whenever possible. Consequently, the staircase join algorithms had to be modified accordingly. The adapted algorithms work similarly to the nested-loop join — more precisely, similarly to the index nested-loop join.

On the basis of the current context tuple's pre value, $pre(outer)$, the staircase join node requests the next inner tuple from the document table. The inner-join index on the document table guarantees that it satisfies the pre clause. If the post clause is satisfied, too, the join is executed and the result tuple returned. After that, the join resumes execution with the retrieval of the next inner tuple. During this process, the executor must make sure that the inner tuple lies within the current partition. If not, it must switch to the next partition by requesting the next context

[5] They will be called *outer* and *inner*, respectively, in the following.

node from the outer subplan. If the outer relation runs out of tuples, the execution of the join is complete.

3.5.3 The Staircase Join's Execution Module

The clearly distinguished execution steps predefined by pipelining and the three staircase join-specific techniques (pruning, partitioning, and skipping) suggest the use of a *state automaton* to implement the staircase join's execution module. For example, there could be one state for requesting the next outer and inner tuple, respectively, one state to evaluate the post clause and another one which is responsible for the creation of the result tuple. Pruning and the test for inclusion in the current partition could also be mapped to their own state. The same approach is used in the execution module of the merge join.

In the course of the implementation, each XPath axis (`preceding`, `following`, `descendant`, and `ancestor`) was assigned its own automaton. Which one of them is called is determined on the top level of staircase join execution illustrated in Algorithm 5. The `exec_strcs_join()` routine is executed in an infinite loop until the execution of the join is complete.

```
exec_strcs_join (plan : StrcsJoin)
BEGIN
    SWITCH plan.type DO
        CASE DESCENDANT
            exec_desc_axis (plan);
        CASE ANCESTOR
            exec_anc_axis (plan);
        CASE FOLLOWING
            exec_fol_axis (plan);
        CASE PRECEDING
            exec_prec_axis (plan);
END
```

Algorithm 5: The top-level routine for executing a staircase join.

The four state automatons themselves run in an infinite loop as well. It is terminated when the next result tuple can be returned to the caller. Within the loop there is a *switch-case* statement which keeps track of the current state. Upon the automaton's next call, execution is resumed in the latest state. The layout of the state automatons is illustrated in Algorithm 6. The routines involved in the execution of the staircase join are listed in Appendix D.2.

3.5.4 Context Pruning, Partitioning, and Skipping

This section describes the steps involved in the implementation of the state automatons of the `preceding`, `following`, `descendant`, and `ancestor` axis.

Preceding and Following Axis

In case of the `preceding` and `following` axis, pruning is the most important concept involved in the join-based evaluation of the location step. Due to the nature of the respective pruning technique, it must be applied in one go before the execution of the join. The `preceding` axis allows for the removal of all context nodes except the one with the highest pre value. However, PostgreSQL does not offer

```
exec_XXX_axis (plan : StrcsJoin)
BEGIN
    FOR EVER DO
        SWITCH plan.state DO
            CASE INITIALIZE
                ...
            CASE TEST_POST
                ...
            CASE ...
                ...
            CASE JOIN
                RETURN result
END
```

Algorithm 6: Layout of a state automaton.

an appropriate means to access the last tuple in a relation directly.[6] Instead, all tuples from the outer subplan must be retrieved sequentially from first to last. A similar technique is applied in case of the `following` axis. While the staircase join plan requests one tuple after another from the outer subplan, it examines their post value and remembers the tuple with the highest value.

The actual joins responsible for the evaluation of the `preceding` and `following` location steps were implemented very similarly to the join algorithms of the other two axes. The main difference is that they work on one partition only. Figure 3.8 shows the partitions scanned for both axes. In case of the `preceding` axis, it contains the document nodes with a pre value between the minimum pre value 0 and $pre(outer) - 1$, where *outer* is the single context node. In case of the `following` axis, they are the document nodes whose pre values lie between $pre(outer) + 1$ and $max(pre(inner))$, where *outer* is the single context node and $max(pre(inner))$ is the maximum pre value in the document table. Thanks to the inner-join index on the document table, the first inner tuple within the partition can be accessed directly.[7] Since the index makes sure that it satisfies the pre clause, we can immediately proceed with the evaluation of the post clause which takes place within the staircase join execution module. If it is found to be satisfied, too, another result tuple can be returned. If no more tuple can be returned by the index, the evaluation of the location step is complete.

Descendant Axis

The pruned context set of the `descendant` axis typically consists of several tuples. As we do not wish to materialize the result of pruning, the removal of redundant tuples must be carried out in interaction with the partition-wise execution of the staircase join.

The state automaton for the `descendant` axis is outlined in Algorithm 7. After the first context tuple has been retrieved from the outer subplan in the `INITIALIZE`

[6] In case of the initial context set *cs*, the last tuple in the relation could be quite easily retrieved by a backwards scan of *cs*. However, this approach was not followed through, because, if the outer relation represented the result of the previous join, it would have involved the materialization of the outer tuples.

[7] Although this is not an advantage in case of the `preceding` axis — the scan of the inner relation must begin at the very first tuple —, it may really make a difference with respect to the `following` axis.

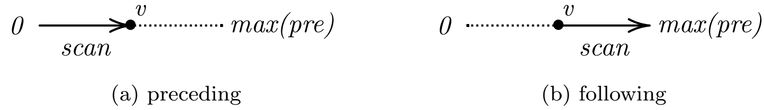


Figure 3.8: The partitions scanned for the `preceding` (a) and `following` (b) axes in the document table based on the pruned context node v. Due to the inner-join index on the document table, the executor can jump directly to the first document node in the partition.

state, it is stored as lower boundary of the first partition, $outer_1$. To identify the upper boundary of the partition ($outer_2$), the `NEXT_OUTER` state prunes all tuples subsequently returned by the outer subplan, until the next one with a higher post value than $outer_1$ is found. As soon as the first partition is set, we can start to retrieve the inner tuples within the partition (see states `RESCAN` and `NEXT_INNER`). To do so, a rescan of the inner-join index is initiated, which makes sure that all returned inner tuples (*inner*) satisfy the pre clause, i.e., that they have a higher pre value than the lower partition boundary $outer_1$. The `TEST_PARTITION` state verifies that the pre value of *inner* does not exceed the upper partition boundary. If the post clause is also satisfied for $outer_1$ and *inner*, the `JOIN` state can build and return another result tuple.

If the `TEST_PARTITION` state encounters the first inner tuple outside the current partition, the executor must switch to the next partition. The last partition is a special case. It lies between the pre value of the last context tuple and the last tuple in the document table. Therefore, the `TEST_PARTITION` state becomes redundant, if the upper partition boundary $outer_2$ evaluates to `NULL`.

The inner-join index on the document table is also responsible for the efficiency of skipping. In case of the `descendant` axis, skipping is incorporated into the `TEST_POST` state. If the post clause is found to be false, we have found the first following node of $outer_1$ and may skip the remaining inner tuples in the current partition.

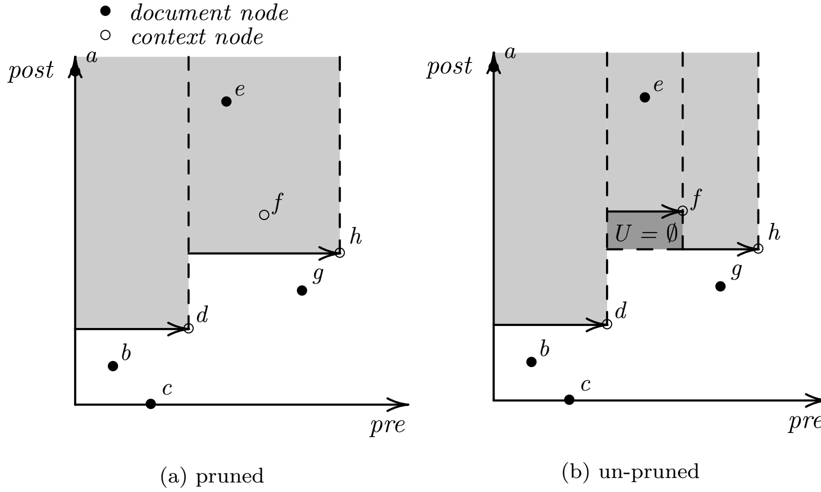
The Ancestor Axis

The `ancestor` axis turned out to be a special case in many respects during the implementation. For once, we decided not to apply pruning, because in PostgreSQL, it does not offer any advantage for the evaluation of the `ancestor` axis. First, it is partitioning, not pruning, which prevents the creation of duplicates. Second, the application of pruning would result in one additional comparison of post values per context set node. If there is a large number of context nodes, this may negatively affect performance, because according to the PostgreSQL source code documentation, one of the critical time factors during execution is the evaluation of expressions (such as the pruning condition).

The correctness of the result obtained by using an *un-pruned* context set is ensured by the empty region behavior observed in the pre/post plane. Figure 3.9 illustrates the situation. If pruning is applied, node f is removed from the context set, because it is an ancestor of h. The gray area in Figure 3.9 (a) depicts the result region obtained, if we evaluated the `ancestor` axis on the basis of a pruned context set. Although it is larger than the result region in Figure 3.9 (b), which is obtained using an un-pruned context set, both regions contain exactly the same tuples. This is because the dark gray area in Figure 3.9 (b) is necessarily empty, as it corresponds to the empty region U as described in Section 2.3.1.

```
exec_desc_axis (plan : StrcsJoin)
BEGIN
  FOR EVER DO
    SWITCH plan.state DO
      CASE INITIALIZE
        /* request the first context node */
        outer2 ← exec_outer_subplan ();
        state ← STORE;
      CASE STORE
        /* set lower partition boundary (outer1) and initiate pruning */
        IF !outer2 THEN
          RETURN null;                /* completion of join */
        outer1 ← outer2;
        state ← NEXT_OUTER;        /* pruning */
      CASE NEXT_OUTER
        /* apply pruning and set upper partition boundary (outer2) */
        outer2 ← exec_outer_subplan ();
        IF eval_expr (post(outer2) > post(outer1)) || !outer2 THEN
          state ← RESCAN;
        /* otherwise remain in this state */
      CASE RESCAN
        /* initiate rescan on the basis of outer1 */
        exec_index_rescan (outer1);
        state ← NEXT_INNER;
      CASE NEXT_INNER
        /* request the next document node */
        inner ← exec_inner_subplan ();
        IF !inner THEN
          RETURN null;                /* completion of join */
        state ← TEST_PARTITION;
      CASE TEST_PARTITION
        /* are we still within partition? */
        IF outer2 THEN
          IF !eval_expr (pre(outer2) > pre(inner)) THEN
            state ← STORE;       /* get next partition */
        state ← TEST_POST;
      CASE TEST_POST
        /* evaluate the post clause */
        IF eval_expr (post(outer1) < post(inner)) THEN
          state ← JOIN;
        ELSE
          state ← STORE;       /* skipping */
      CASE JOIN
        /* execute the join */
        state ← NEXT_INNER;
        RETURN exec_join_project (outer1, inner);
END
```

Algorithm 7: Outline of the state automaton for the `descendant` axis. $outer_1$ is the lower partition boundary and $outer_2$ the upper partition boundary. *inner* is the currently considered document node.



(a) pruned

(b) un-pruned

Figure 3.9: The result region of ancestor axis evaluation with (a) and without pruning the context set. Although the result region in (a) is larger than in (b), both regions produce exactly the same tuples, because of empty region U.

Another difference of the `ancestor` axis is that the staircase join — and not the index — is responsible for the evaluation of the pre clause. This is due to the nature of the clause: $pre(outer) > pre(inner)$. If it was used to initialize index rescans on the basis of *outer*, each rescan would begin at the very first document node and produce a lot of duplicate result tuples. So instead of the pre clause, the index was provided with a new predicate which enables the implementation of skipping. Figure 3.10 shows a staircase join plan for an ancestor join between cs and doc_1.

Algorithm 8 gives an outline of the state automaton for the `ancestor` axis. The partitions considered for this axis lie between the pre values of two un-pruned context tuples. However, there is no need to physically store the lower partition boundary here. We must only keep record of the upper boundary, *outer*, which also represents the currently considered context node. The pre clause makes sure that the requested inner nodes have a lower pre value than this boundary (see `TEST_PRE` state). If it is no longer satisfied, we have reached the inner tuple with $pre(outer) = pre(inner)$ and must switch to the next partition. To do so, the executor may now simply proceed with the retrieval of the next context tuple from the outer relation in the `NEXT_OUTER` state. *inner* is guaranteed to be the first node within the new partition, i.e., there is no need for the additional evaluation of a lower partition boundary and we may continue directly with the evaluation of the pre clause.

If the `TEST_PRE` state finds that the current inner tuple satisfies the pre clause, the algorithm proceeds with the evaluation of the post clause in the `TEST_POST` state. If it is also satisfied, the `JOIN` state may build and return the next result tuple.

Skipping. The implementation of skipping also required a few special provisions in case of the `ancestor` axis. It has already been indicated that a new clause was incorporated into the inner-join index to enable the application of this technique. In Section 2.3.3, it was said that the *number* of nodes that may be skipped during `ancestor` axis evaluation amounts to $skipnum = post(n) - pre(n)$ (n is the document node which initiates the skipping process). To determine the destination of skipping, *skipnum* must be added to the pre value of n. Thus, we may skip to

```
exec_anc_axis (plan : StrcsJoin)
BEGIN
    FOR EVER DO
        SWITCH plan.state DO
            CASE INITIALIZE
                /* request the first context node */
                outer ← exec_outer_subplan ();
                skip_tup ← (tuple) 0;
                state ← RESCAN;
            CASE RESCAN
                /* initiate rescan on the basis of skip_tup */
                exec_index_rescan (skip_tup);
                state ← NEXT_INNER;
            CASE NEXT_INNER
                /* request the next document node */
                inner ← exec_inner_subplan ();
                IF !inner THEN
                    RETURN null;                    /* completion of join */
                state ← TEST_PRE;
            CASE TEST_PRE
                /* evaluate the pre clause */
                IF !outer THEN
                    RETURN null;                    /* completion of join */
                IF eval_expr (pre(outer) > pre(inner)) THEN
                    state ← TEST_POST;
                ELSE
                    state ← NEXT_OUTER;        /* next partition */
            CASE TEST_POST
                /* evaluate the post clause */
                IF eval_expr (post(outer) < post(inner)) THEN
                    state ← JOIN;
                ELSE
                    IF eval_expr (post(inner) > pre(inner)) THEN
                        skip_tup ← inner;            /* skipping*/
                    state ← RESCAN;
            CASE JOIN
                /* execute the join */
                state ← NEXT_INNER;
                RETURN exec_join_project (outer, inner);
            CASE NEXT_OUTER
                /* switch to the next partition */
                outer ← exec_outer_subplan ();
                state ← TEST_PRE;
END
```

Algorithm 8: Outline of the state automaton for the `ancestor` axis.

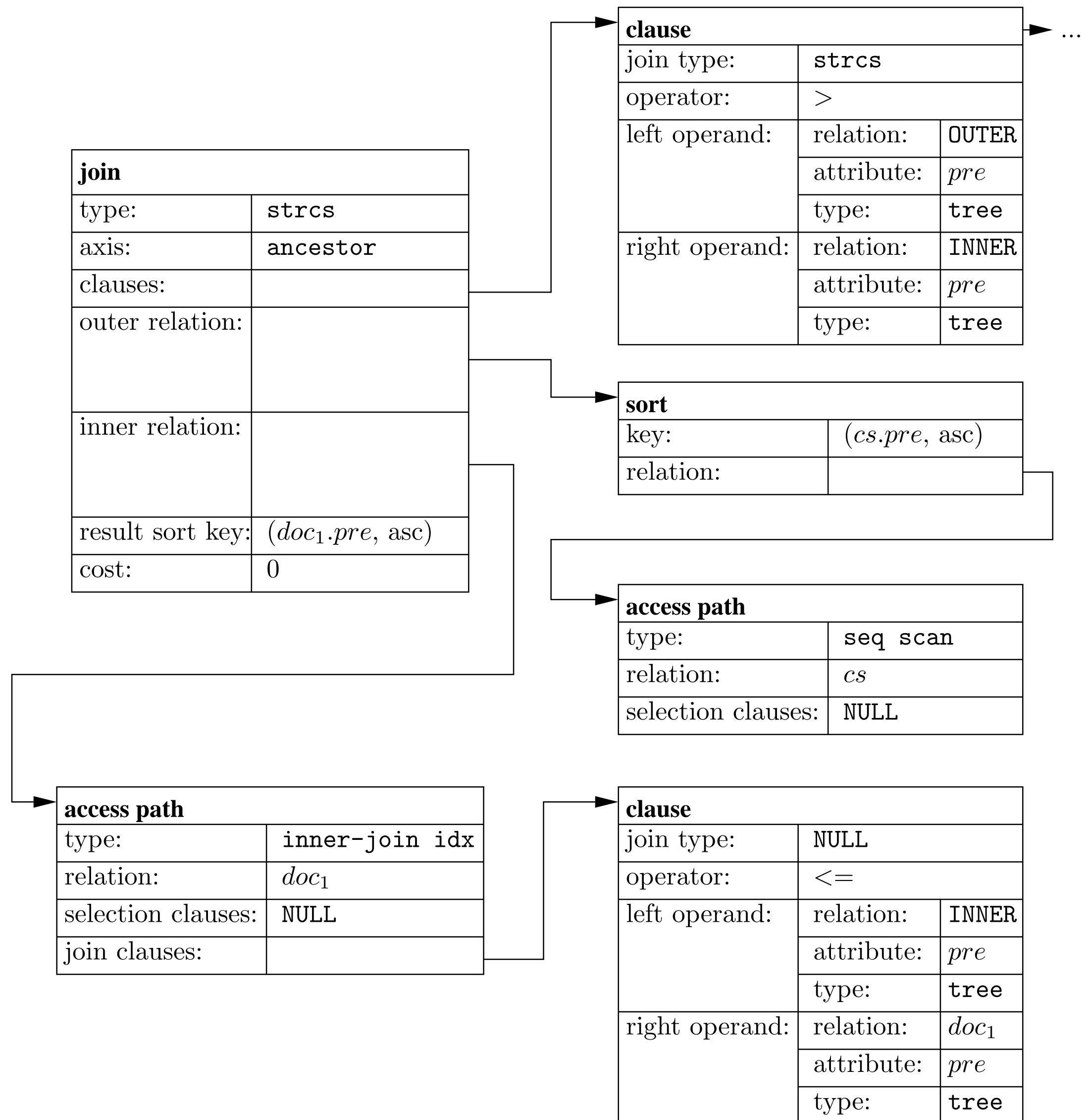


Figure 3.10: The execution plan created for an ancestor join between *cs* and doc_1. The pre clause was *not* removed from the list of join clauses and the inner-join index was provided with a new clause which enables skipping.

the node with a pre value of:

$$pre_{skip} = pre(n) + skipnum = pre(n) + (post(n) - pre(n)) = post(n).$$

This piece of knowledge was incorporated into the new index clause of the inner-join index. Figure 3.10 shows that it allows us to use the post value of an inner tuple as search key in the index rescan of the document table. All tuples returned by the index have a pre value which is equal to or larger than the search key's post value.

Just as in case of the `descendant` axis, skipping was incorporated into the `TEST_POST` state. It is initiated, if the post clause is no longer satisfied. The additional predicate $post(inner) > pre(inner)$ is required to avoid that skipping is directed backwards.

3.5.5 Completing Staircase Join Execution

The third recursive descent in the plan tree after execution is finished is dedicated to a general clean-up, e.g. for the deallocation of memory used by the tuple table slots. The routine responsible for the completion of the staircase join can be found in Appendix D.3.

3.5.6 Further Implementation Considerations

The importance of the inner-join index for an efficient execution of the staircase join was already stressed in various places. It is the reason why it was tried to re-locate the evaluation of additional clauses into the index. Candidates are the evaluation of the upper partition boundary in case of the `descendant` axis and the pre clause in case of the `ancestor` axis.

Unfortunately, this idea failed due to limitations of PostgreSQL's clause evaluation mechanisms. It would have implied the possibility to evaluate two index clauses on the basis of two distinct tuples. In case of the `descendant` axis, the index would have to evaluate the pre clause on the basis of $outer_1$ and the partition clause on the basis of $outer_2$. In case of the `ancestor` axis, it would have involved the use of *skip_tup* in the evaluation of the skipping condition and the use of *outer* in the evaluation of the pre clause.

Chapter 4

Performance Tests

This chapter is dedicated to the performance tests that were carried out in both, a tree-aware and an unmodified instance of PostgreSQL. In both databases, the buffer-related behavior and the execution times of all four axes (`descendant`, `ancestor`, `preceding`, and `following`) were examined in dependence on the size of the document table.

4.1 The Test Environment

The performance tests were executed on a 2.2 GHz Dual-Pentium 4 machine with 2 GB RAM, on which two independent instances of PostgreSQL 7.3.3. were installed: the unmodified version as downloaded from the web site of the PostgreSQL Global Development Group and the tree-aware, staircase join-enhanced version.

4.1.1 Creation of Test Documents

Both database instances were supplied with six document tables of increasing size. They originate from the XML document generator *XMLgen*, developed for the *XMark benchmark project* [SWK+02]. It can create XML documents of varying, user-defined size which are based on the DTD of an internet auction site (see Appendix E). The size and the number of nodes contained in the generated documents are listed in Table 4.1. The height of all created XML trees is 11.

	XML document size	|nodes|
A	110 KB	5,257
C	1.1 MB	52,180
D	11 MB	511,474
E	58 MB	2,538,027
F	110 MB	5,077,531
G	1.1 GB	50,894,789

Table 4.1: Characteristics of the XML documents generated for the performance tests.

4.1.2 Loading the XML Documents

The data contained in the XML documents were extracted in a sequential, SAX-based parsing pass and then loaded into PostgreSQL. During parsing, each document node was assigned its preorder and postorder value and the pre value of its

parent. This process was not only applied to element nodes, but also extended to attributes and text content. Apart from these values, the extracted information comprise a numeric identifier, which specifies whether a node represents an XML element (0), text content (1), or an attribute (2), and the name of the element or attribute or `NULL` in case of a text content node.

The extracted data were loaded into PostgreSQL with the SQL `COPY` command [WD02], which offers the possibility to load any amount of table rows in one transaction. Both, context set and document table, have the same schema and data types. In the tree-aware PostgreSQL instance, they are as follows:

(*pre* `tree unique`, *post* `tree unique`, *par* `tree`, *kind* `int`, *name* `varchar(10)`).

In the original instance, they read:

(*pre* `int unique`, *post* `int unique`, *par* `int`, *kind* `int`, *name* `varchar(10)`).

After the tables were loaded, indices were created on the document tables. They have the form of B-trees and differ slightly for the modified and unmodified PostgreSQL instance. In the tree-aware database, they involve the attributes *pre*, *kind*, and *name*. In the original database, the *post* column was additionally included. Before the tests were started, the SQL command `ANALYZE` [Pos02] was sent to the database. It initiates the collection of statistics which are used by the planner/optimizer to provide better execution plans.[8]

4.1.3 Test Queries

The four XPath expressions that were translated into SQL test queries consist of two location steps. All of them start off at the root node, i.e., the initial context set contains one tuple only. The first step, a `descendant` step, is intended to create a reasonable context set for the second step whose behavior is of actual interest to us.

Q_{desc}: `//descendant::open_auction/descendant::description`
Q_{anc}: `//descendant::age/ancestor::person`
Q_{prec}: `//descendant::current/preceding::initial`
Q_{fol}: `//descendant::city/following::zipcode`

The XPath expressions were translated into SQL queries according to the principles described in Section 2.2.2. For example, the SQL statement for Q_{anc} looks as follows:

```
SELECT DISTINCT doc2.*
  FROM context cs, document doc1, document doc2
 WHERE cs.pre < doc1.pre AND cs.post > doc1.post          -- descendant
   AND doc1.kind = 0 AND doc1.name = 'age'
   AND doc1.pre > doc2.pre AND doc1.post < doc2.post      -- ancestor
   AND doc2.kind = 0 AND doc2.name = 'person'
 ORDER BY doc2.pre;
```

For each query, two independent test series were carried out on both database instances. The purpose of the first series was to inspect the final execution plan and to obtain statistics about database activity, e.g. about the number of disk blocks requested from a table or index and the number of buffer hits herein. The second series was exclusively dedicated to the measurement of the queries' execution times.

[8] It turned out that the unmodified PostgreSQL instance chooses inferior plans for queries on documents A and C, if no statistics are previously collected.

Shrink-Wrapping the Descendant Axis

To obtain optimal test results in the original database, an additional optimization called *shrink-wrapping* was applied in this PostgreSQL instance. It is a means to enhance the tree-awareness of an RDBMS on the SQL-level and only applicable to the `descendant` axis.

Shrink-wrapping limits the search for the descendants of a context node v to the XML subtree rooted at v. Such a subtree can be defined in terms of two intervals of pre and post values, i.e., a node n is part of the subtree, if its pre and post value lie within the following boundaries:

$$pre(v) < pre(n) <= post(v) + h(t) \quad \wedge \quad pre(v) - h(t) <= post(n) < post(v)$$

(where $h(t)$ is the height of the XML document tree). More details on how these intervals are derived can be found in Section 5 of [Gru02]. The intervals can be converted into the following join clauses:

$$\texttt{WHERE}\ cs.pre < doc_1.pre\ \texttt{AND}\ cs.post > doc_1.post$$
$$\texttt{AND}\ doc_1.pre <= cs.post + 11\ \texttt{AND}\ doc_1.post >= cs.pre - 11$$

Note that the first two of them correspond to the pre and post clause of the `descendant` axis such that only the other two are additionally included into the queries, if a `descendant` axis is present. Since all XML trees in the performance tests have a height of 11, $h(t)$ has been set accordingly.

4.1.4 Execution Plans

Figure 4.1 shows the execution plan of the test queries as chosen by the tree-aware PostgreSQL instance. It looks the same for all four queries and all document sizes. The difference between the plans lies in the location of clause evaluation. In case of a `descendant`, `preceding`, and `following` join, the index scan on the document table evaluates three clauses, namely the pre clause and the two selection clauses on the *kind* and *name* attribute. The staircase join node itself processes a further join clause, namely the post clause. In case of an `ancestor` location step, the staircase join node evaluates the pre and the post clause, while the index is responsible for the two selections and the additionally introduced skipping condition.

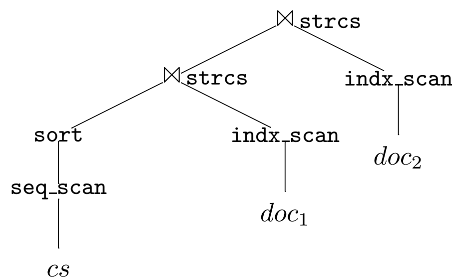


Figure 4.1: The execution plan of the test queries in the tree-aware PostgreSQL instance for documents A to G (index on *doc.pre*, *doc.kind*, *doc.name*).

The unmodified PostgreSQL instance uses two consecutive nested-loop joins to answer the test queries. The corresponding plan tree is illustrated in Figure 4.2. Independently of the XPath axis, all clauses are evaluated during the index scan of the document table. In case of the `descendant` axis, these are the pre and the post clause, the two shrink-wrapping clauses, and the two selection clauses on the *kind*

and *name* attribute. In case of the other three axes, the index evaluates the pre and the post clause and the two selection clauses.

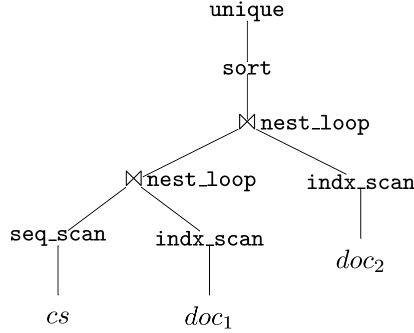


Figure 4.2: The execution plan of the test queries in the original PostgreSQL instance for documents A to G (index on *doc.pre*, *doc.post*, *doc.kind*, *doc.name*).

4.2 Evaluation of Test Results

To make the test results comparable, it was ensured that the factors that might influence them grow linearly. First, there is the size of the considered XML documents whose growth is recorded in Table 4.1. However, since we have found a way to circumvent a sequential scan of the document table and can jump directly to a required tuple, it is not so much the document size that is decisive. What is more important is the subset of index entries that must effectively be accessed during the index scan of the document table. In the worst case, it amounts to a scan of all index entries, e.g., if we evaluate a `descendant` location step on the basis of the root node. As for the test queries, XMLgen turned out to make provisions for this issue. It ensures that the subtrees in the generated XML documents grow linearly to the size of the documents themselves such that suitable queries were easy to find. Because of this correspondence between the document size and the number of effectively scanned tuples, both factors will be summarized under the term *document size* in the following.

The second factor that may affect the performance behavior is the size of the context set (either the initial context set or the number of result tuples returned by the previous location step). Table 4.2 shows that the result set also grows linearly to document size.

	Q_{desc}	Q_{anc}	Q_{prec}	Q_{fol}
A (110 KB)	12	8	12	12
C (1.1 MB)	120	77	120	125
D (11 MB)	1200	631	1200	1255
E (58 MB)	6000	3163	6000	6359
F (110 MB)	12,000	6409	12,000	12,715
G (1.1 GB)	120,000	64,463	120,000	127,315

Table 4.2: Result sizes obtained for the test queries in the performance tests. Both location steps return an identical number of tuples.

4.2.1 Disk Page Loads and Buffer Hits

This section examines a number of buffer-related aspects observed during the execution of the test queries. By default, the PostgreSQL buffer cache can hold 64 disk blocks. Each of them has a size of 8 KB.

The initial context set table *cs* contains one tuple only. Its buffer behavior is constant over all queries, all XML documents and both database instances. It occupies one disk page which is loaded once at the very beginning of execution and never requested a second time. For the evaluation of the test results, it is negligible. The same applies to the intermediate context set, i.e. the result of the previous location step. Due to pipelining, it has no effect at all on the buffer statistics.

As a result, the following sections may entirely focus on the buffer statistics of the document tables and their associated indices. In both categories, we will examine the number of pages that must be loaded from disk and the number of buffer hits, i.e. the number of requested pages that are already available in the buffer.

Note that PostgreSQL only requests a tuple from the document table, if it was found to satisfy all index clauses during the index scan.

Buffer Behavior in the Tree-Aware Database

Table 4.3 and Figure 4.3 illustrate the buffer-related behavior of the test queries in the tree-aware PostgreSQL instance. In addition to the absolute number of page loads and buffer hits, Table 4.3 also displays the percentage of buffer hits for document table and index.

Buffer statistics of the index. The statistics show that the *total* number of disk pages requested from the index grows linearly to the document size for all test queries. This is hardly surprising, because the evaluation of each location step boils down to a single index-based scan of the document table. However, there is a wide discrepancy between Q_{desc} and Q_{anc} on the one hand and Q_{prec} and Q_{fol} on the other, both, with respect to the absolute number of requested index pages and the percentage of buffer hits.

The considerable number of buffer hits for Q_{desc} and Q_{anc} is caused to a large extent by partitioning and the interconnected manner in which the location steps of the test queries are executed. The `descendant` and `ancestor` algorithms typically work on several partitions (as opposed to the single partition considered in case of the `preceding` and `following` axis). When the algorithms switch to the next partition, the tuples that make up the partition's boundaries — and hence the pages on which they are situated — are already in the buffer cache, because they were loaded immediately beforehand to execute the outer subplan. Thus, when the algorithms start to work on the tuples within the partition, it is very likely that they reside on a disk page already in the buffer. Another influential factor is the physical closeness of the requested tuples. It increases the probability that the content of two successive partitions resides on the same buffer page.

In contrast to Q_{desc} and Q_{anc}, the number of index buffer hits for Q_{prec} and Q_{fol} amounts to 0 for all documents (except A). This is because the two location steps of Q_{prec} and Q_{fol} are evaluated independently of each other due to the pruning technique applied here. Before the `preceding` and `following` algorithms can determine which node is the single context node, they must have requested all tuples from their outer subplan. Only after that, they begin with the actual evaluation of the location step. This means that there are two totally independent scans of the document table index, one to execute the first location step (i.e. the `descendant` axis) and one to execute the second step (the `preceding` or `following` axis). The index buffer hits observed for document A are caused by the fact that the whole

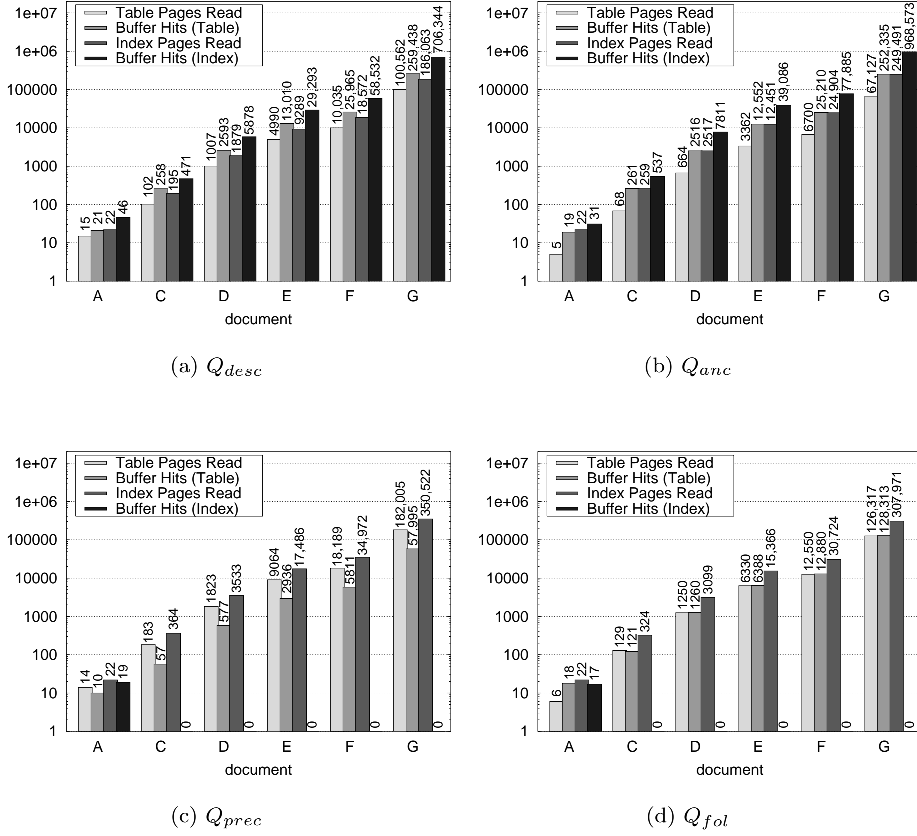
Q_{desc}	# loads (index)	# hits (index)	% of hits	# loads (table)	# hits (table)	% of hits
A	22	46	67	15	21	58
C	195	471	70	102	258	71
D	1879	5878	75	1007	2593	72
E	9289	29,293	75	4990	13,010	72
F	18,572	58,532	75	10,035	25,965	72
G	186,063	706,344	79	100,562	259,438	72
Q_{anc}						
A	22	31	58	5	19	79
C	259	537	67	68	261	79
D	2517	7811	75	664	2516	79
E	12,451	39,086	75	3362	12,552	78
F	24,904	77,885	75	6700	25,210	79
G	249,491	968,573	79	67,127	252,335	78
Q_{prec}						
A	22	19	46	14	10	41
C	364	0	0	183	57	23
D	3533	0	0	1823	577	24
E	17,486	0	0	9064	2936	24
F	34,972	0	0	18,189	5811	24
G	350,522	0	0	182,005	57,995	24
Q_{fol}						
A	22	17	43	6	18	75
C	324	0	0	129	121	48
D	3099	0	0	1250	1260	50
E	15,366	0	0	6330	6388	50
F	30,724	0	0	12,550	12,880	50
G	307,971	0	0	126,317	128,313	50

Table 4.3: Buffer-related behavior of the test queries in the tree-aware PostgreSQL instance. In addition to the number of page loads and buffer hits (document table and index) the percentage of buffer hits is displayed.

index fits into the buffer.

The difference in the absolute number of requested index pages between the four queries is caused by the number of required index rescans. While the `preceding` and `following` axes require exactly one index rescan, there are typically several in case of the `descendant` and `ancestor` axes (due to partitioning and skipping) and each of them involves another descent in the B-tree index. A further factor might be the number and distribution of the scanned index entries over the document table. The more index entries must be scanned or the wider they are spread, the more index pages must be loaded.

Buffer statistics of the document table. With respect to the total number of requested document pages, the statistics show the same linear trend as the index. The request for document pages is strictly dependent on the number of index entries that satisfy the index clauses, because only then the respective document tuple is retrieved.



(a) Q_{desc} (b) Q_{anc}

(c) Q_{prec} (d) Q_{fol}

Figure 4.3: Buffer-related behavior of the test queries in the tree-aware PostgreSQL instance. The total number of pages requested from the document table index and the document table itself grows linearly to the document size.

Once again, Q_{desc} and Q_{anc} show a higher percentage of buffer hits than Q_{prec} and Q_{fol} in this category. Just as described before, this development is due to the interconnected/independent execution of the queries' location steps. The buffer hits for Q_{prec} and Q_{fol} are a result of the physical closeness of the result tuples.

Buffer Behavior in the Original Database

Table 4.4 and Figure 4.4 illustrate the buffer-related behavior of the test queries in the unmodified PostgreSQL instance. Table 4.4 additionally displays the percentage of buffer hits for document table and index.

Q_{desc} shows very similar statistics to its counterpart in the tree-aware database. Not only do the measured values of page requests and buffer hits grow linearly to the size of the document table, but there is also a similarity with respect to the absolute numbers and, at least for the index, a similarity in the percentage of buffer hits. These observations indicate that the shrink-wrapping clauses provide extremely precise estimations for the `descendant` axis. Without them, Q_{desc} showed similar statistics to the other three queries tested here.

One thing that becomes apparent at closer inspection of Q_{desc} is that the *total* number of pages requested from the document table is lower than in the tree-aware database (actually, it is reflected in the number of buffer hits). The reason for this

Q_{desc}	# loads (index)	# hits (index)	% of hits	# loads (table)	# hits (table)	% of hits
A	25	45	64	14	10	41
C	227	556	71	101	139	57
D	2157	5576	72	1006	1394	58
E	10,679	27,807	72	4990	7010	58
F	21,358	55,603	72	10,035	13,965	58
G	214,042	676,419	75	100,561	139,439	58
Q_{anc}						
A	25	87	77	5	11	68
C	7729	77	0.9	62	92	59
D	605,887	631	0.1	626	636	50
E	15,034,048	3163	0.02	3106	3220	50
Q_{prec}						
A	25	215	89	14	76	84
C	19,012	120	0.6	5576	1804	24
D	1,852,235	1200	0.06	548,067	173,733	24
E	45,965,699	6000	0.01	13,540,120	4,468,880	24
Q_{fol}						
A	25	199	88	6	84	93
C	16,182	125	0.7	3864	4136	51
D	1,503,040	1257	0.08	394,301	395,094	50
E	37,552,045	6385	0.01	10,075,462	10,152,517	50

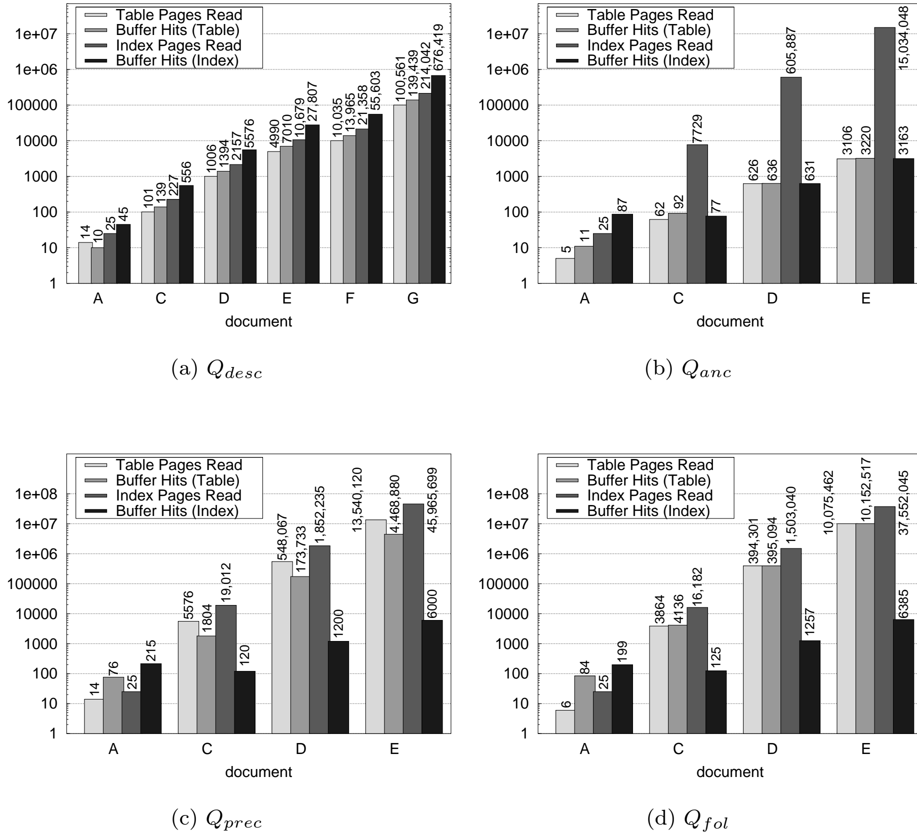
Table 4.4: Buffer-related behavior of the test queries in the original PostgreSQL instance. In addition to the number of page loads and buffer hits (document table and index) the percentage of buffer hits is displayed.

observation is that the original database chooses an execution plan in which all clauses are evaluated in the index. As a result, there is no redundancy with respect to the requested document tuples. A tuple is only retrieved from the document table, if all clauses are found to be satisfied, i.e., all requested tuples are indeed part of the join result. In contrast to that, the staircase join evaluates the post clause outside the index. As a consequence, some tuples may be requested "in vain". Another effect of the reduced request for document tuples is that the physical distance between the requested tuples becomes wider. This is probably the reason for the decreased percentage of buffer hits with respect to the document table.

The other three queries perform considerably inferior than Q_{desc}.[9] However, there are differences in the deterioration, especially between Q_{anc}, on the one hand, and Q_{prec} and Q_{fol}, on the other.

Buffer statistics of the index. Q_{anc}, Q_{prec}, and Q_{fol} show a sharp increase in the total number of requested index pages. It is caused by the evaluation mechanisms of the nested-loop join. In contrast to the staircase join, the nested-loop join re-scans the document table once per outer tuple. Thus, there are two factors that influence the request for index pages: the size of the context set and the document size. Since both grow according to the same factor, the number of requested index pages

[9] In fact, the performance of Q_{anc}, Q_{prec}, and Q_{fol} degraded to such an extent that time constraints made it impossible to execute them on documents F and G (see also Section 4.2.2).



(a) Q_{desc} (b) Q_{anc}

(c) Q_{prec} (d) Q_{fol}

Figure 4.4: Buffer-related behavior of the test queries in the original PostgreSQL instance. While shrink-wrapping causes the total number of pages requested from the index and the document table to grow linearly to the document size in case of Q_{desc}, the other three queries show a substantial deterioration.

should grow roughly quadratically to the document size. The statistics confirm this assumption.

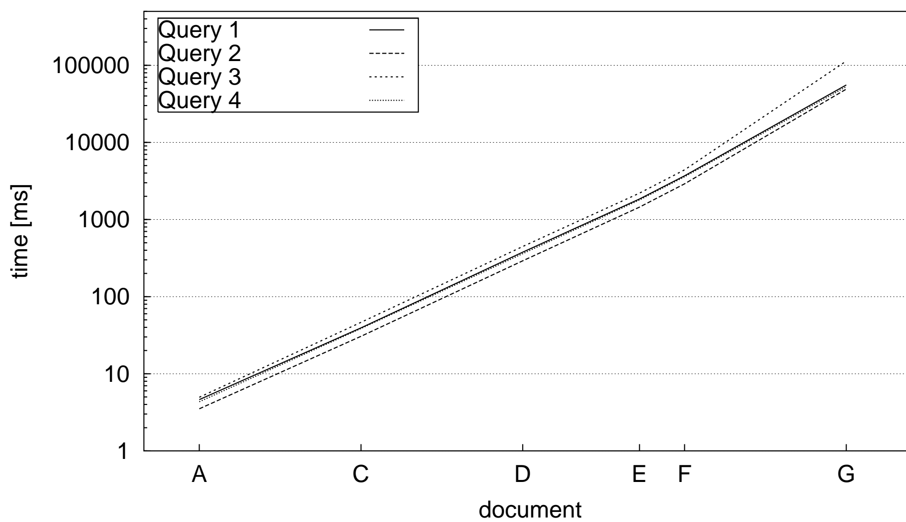
The low number of buffer hits is caused by the numerous index rescans (one per outer tuple) and the distance between their starting and ending positions. In other words, when index rescan i is finished, it is very likely that the first index page requested by index rescan $i+1$ has already been replaced by another page in the buffer.

Buffer statistics of the document table. For Q_{prec} and Q_{fol}, the growth of requested document pages is also roughly quadratic to the document size, while it remains linear in case of Q_{anc}. The reason for this discrepancy lies in the nature of the evaluated axes. The number of a node's ancestors (and descendants for that matter) is restricted by the height of the document tree which typically remains constant in spite of an increasing document size. Applied to Q_{anc}, this means that the index-based evaluation of the region queries results in at most 10 nodes *per node in the context set*. Only these must effectively be retrieved from the document table. The node tests on the *kind* and *name* attribute further reduce this number. The linear growth of document page requests in case of Q_{anc} confirm this reasoning. The decisive factor responsible for this behavior is the number of context set nodes.

In contrast to that, the number of a node's preceding and following nodes is restricted by the number of nodes in the XML document. As a result, the request for document pages in case of Q_{prec} and Q_{fol} is not only dependent on the size of the context set, but also on the size of the document, which in turn accounts for the quadratic growth of document page requests.

4.2.2 Execution Times

This section examines the execution times measured in the two PostgreSQL instances. To calculate the average evaluation time, each query was executed five times. As preliminary measurements showed that the initial execution of a query may take considerably longer (up to twice as long) than any subsequent execution of the same query, an additional execution pass was carried out beforehand to "warm up" caches.



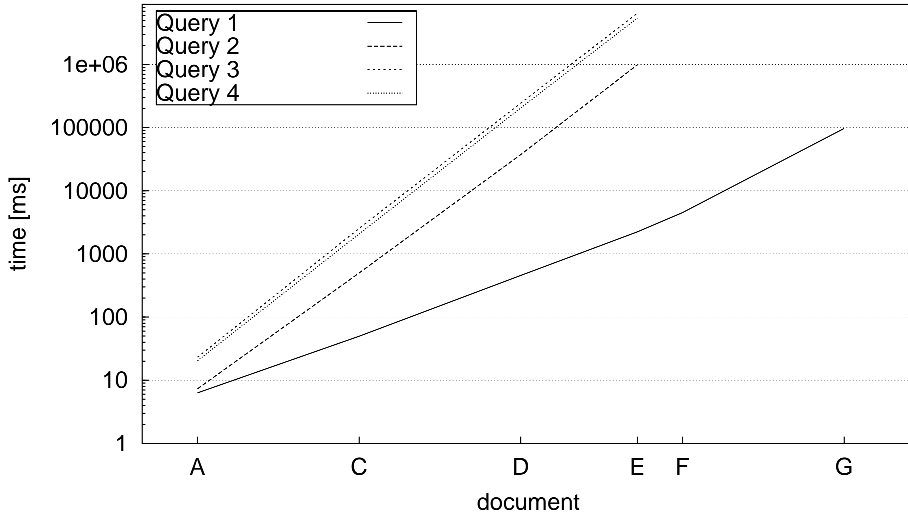
	Q_{desc} (sec)	Q_{anc} (sec)	Q_{prec} (sec)	Q_{fol} (sec)
A	0.0046	0.0035	0.0049	0.0043
C	0.0393	0.0305	0.0464	0.0385
D	0.3776	0.2917	0.4472	0.3613
E	1.8440	1.4474	2.2009	1.7913
F	3.6848	2.8988	4.4041	3.5839
G	55.6422	49.0274	113.5450	52.8533

Figure 4.5: Execution times of the test queries in the tree-aware PostgreSQL instance. Apart from document G, they grow linearly to the document size.

Figure 4.5 illustrates the test results measured in the tree-aware database. They confirm the observations made during the evaluation of the buffer statistics. For all documents except G, the execution times of the test queries grow linearly to the size of the document table. Additional tests, in which both location steps were tuned to produce only one result node, proved that the critical factor is indeed the document size or more precisely, the size of the effectively scanned subset of index entries. Although a slight decrease in the absolute execution times could be

observed in these tests, the measured times still showed the same linear growth.

The deviation of the execution times with respect to document G might be due to caching effects of the operating system. The operating system will try to cache accesses to the database file. However, the table size of document G and the size of its associated index probably exceed the main memory of our test computer.



	Q_{desc} (sec)	Q_{anc} (sec)	Q_{prec} (sec)	Q_{fol} (sec)
A	0.0062	0.0072	0.0231	0.0203
C	0.0497	0.4999	2.5388	2.0757
D	0.4559	37.4523	243.9718	205.3002
E	2.2368	992.9138	6528.1853	5443.9523
F	4.5208	-	-	-
G	97.2034	-	-	-

Figure 4.6: Execution times of the test queries in the unmodified PostgreSQL instance. Those of Q_{desc} grow linearly to the size of the document, while those of the other three test queries increase quadratically to this factor.

Figure 4.6 displays the test results obtained in the unmodified database. Again, they confirm the buffer-related observations previously made for this PostgreSQL instance. Q_{desc} shows very similar execution times to its counterpart in the tree-aware database. For all documents except G, they grow linearly to the size of the document table. The deviation with respect to document G is probably due to the same OS-related reasons as described for the tree-aware instance.

In comparison to Q_{desc}, the other three queries show a substantial deterioration in terms of execution times. In fact, for documents F and G, the staircase join turned out to be the only reasonable alternative capable of handling a join between such amounts of data. In the previous section, this development was attributed to the evaluation mechanisms of the nested-loop join. Since it rescans the document table once per context tuple, we identified two influential factors during the examination of the buffer-related statistics: the growth factors of the context set and of the document table. The measured execution times confirm this. They grow according to the product of these two factors, i.e. roughly quadratically to the doc-

ument size. Additional tests with only one intermediate result tuple confirmed this interpretation. The execution times obtained in these tests showed a linear growth to the size of the document.

Chapter 5

Conclusion and Outlook

The staircase join enhances the tree awareness of an RDBMS and is designed to speed up the SQL-based evaluation of XPath expressions. In the SQL translation of such an expression, each location step (`descendant`, `ancestor`, `preceding`, or `following`) is transformed into a join between the location step's context set and the document table (i.e. the table that stores the data from the queried XML document). The join is governed by two join clauses, the pre and the post clause, which are based on the data provided by the XPath accelerator.

5.1 The PostgreSQL Backend

In the course of this thesis, the staircase join was incorporated into PostgreSQL, an open-source relational database management system. This involved local changes to three out of four query processing stages, namely the parser, the planner/optimizer, and the executor.

The parser was supplied with a new data type `tree`. It is intended to indicate that a column contains tree-specific data and can be assigned to table columns that contain pre or post values. Since these values actually are integer values, the new type was derived from PostgreSQL's `int` type.

The main task in the planner/optimizer was to decide whether the staircase join can be used for the execution of a particular join. First and foremost, the decision depends on the presence of the pre and post clause. If they can both be identified, the staircase join is considered as join operator during dynamic programming. In comparison to the other three native join algorithms of PostgreSQL (nested-loop, merge, and hash join), the staircase join shows two important differences in the context of planning and optimization. First, due to the lack of an appropriate cost function, the join order of an SQL-based XPath query is predefined by the order of location steps in the original XPath expression. Second, for the staircase join, it is sufficient to take only one type of linear join trees into account. This is because both types of trees (left-deep and right-deep) produce exactly the same execution cost, because, internally, both would be executed in exactly the same manner. To bushy trees, the staircase join is not applicable at all. Figure 5.1 shows a typical join tree/execution plan for an SQL-based XPath query with two location steps.

As was indicated before, the choice of the optimal execution plan in the planner/optimizer should ultimately depend on estimations of the execution cost. However, as it would have been beyond the scope of this thesis to write a cost function for the staircase join, its cost were simply estimated to amount to 0. When the performance tests were carried out, this made sure that the staircase join was integrated into the final execution plan.

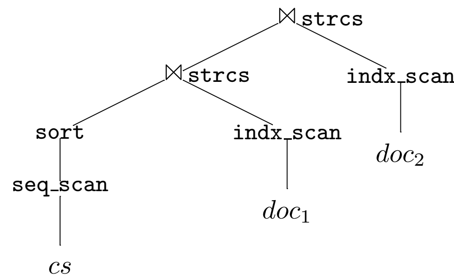


Figure 5.1: The execution plan of an SQL-based XPath query with two location steps. Only left-deep trees are created which means that the context set is always the left or outer relation and the queried XML document table is the right or inner relation.

The work in the executor involved the creation of a totally new module, which is responsible for the execution of the staircase join. It encapsulates implementations of the three essential concepts behind the new join algorithm, namely pruning, partitioning, and skipping. The pipelining mechanisms of PostgreSQL were also an important factor during the implementation. Since the materialization of the join result, i.e. the (intermediate) context set, was to be avoided in PostgreSQL, the context nodes can only be accessed in the sequential manner in which they are produced. Of course, this does mainly affect pruning. For all axes, it means that every single context node has to be handled during pruning. For the `descendant` axis, it also means that pruning has to be carried out in interaction with join execution. In case of the `ancestor` axis, this technique was not applied at all, because it was found not to offer any advantages in PostgreSQL.

As far as the second input parameter of the staircase join, the document table, is concerned, it became clear that an index on that relation is essential for the efficient implementation of partitioning and skipping in PostgreSQL. Thus, the choice of a staircase join in the planner/executor was tied to the availability of such an index.

In the course of the work in the executor, another difference between the staircase join and PostgreSQL's native join algorithms became apparent. During execution, the pre and the post clause must be strictly distinguished and evaluated separately, because the result of their evaluation controls a number of processes in the staircase join's execution module. For example, if the pre clause is no longer satisfied, it is a sign that the execution of a preceding join is complete. In case of an ancestor join, it is an indication that the next partition must be set. Similarly to that, skipping is tied to the evaluation of the post clause. Both, in case of the `descendant` and the `ancestor` axis, it is applied when the post clause is found to be false.

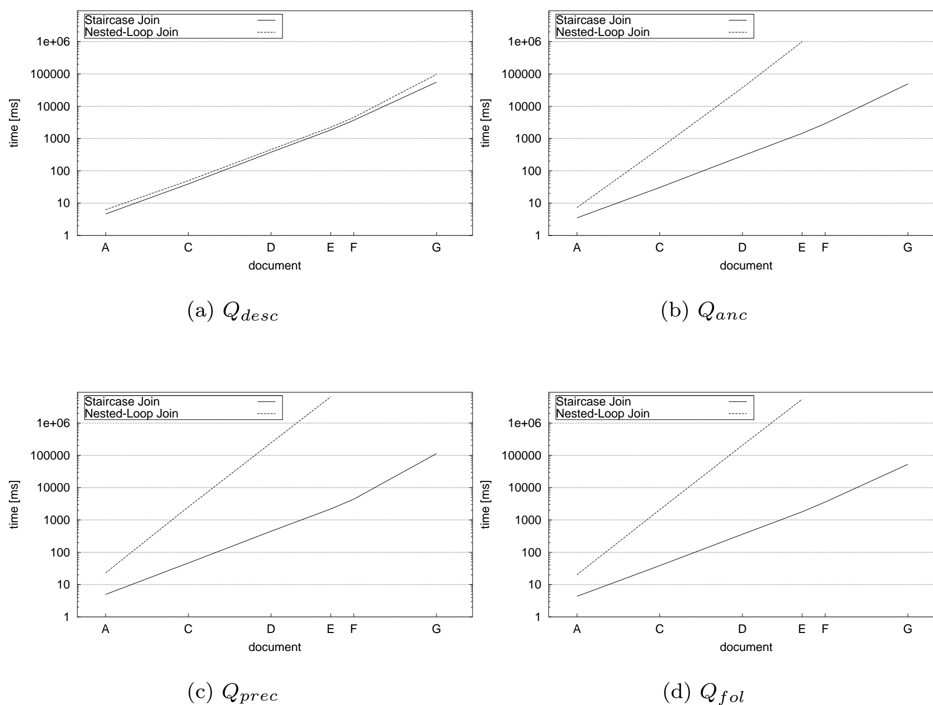
This particularity has the effect that we cannot make optimal use of the document table index. PostgreSQL's index scan implementation does not allow us to distinguish between the evaluation results of two separate index clauses. Either all clauses are satisfied, which means that the next tuple can be returned to the caller, or at least one clause evaluates to false, which means that the scan must be continued. As a consequence, only one of the staircase join clauses, the pre clause, can be evaluated during the index scan of the document table. If the evaluation of the post clause was incorporated into the index, too, essential components of the staircase join, such as partitioning and skipping, could not be applied.

On the other hand, tests have shown that the evaluation of additional clauses in the index may lead to a considerable performance speed-up. A possible solution for this dilemma would be the modification of PostgreSQL's index scan implementation

or even better, the introduction of a new index scan module, especially geared towards the staircase join. Partitioning and skipping could then be re-located into the index module which would in turn allow for the incorporation of the post clause into the index. This approach could also solve the problem described in Section 3.5.6. At present, index clauses can be evaluated on the basis of one reference tuple. However, since the staircase join typically works on a partition between two tuples, the possibility of specifying *two* reference tuples would be preferable.

5.2 The Test Results

In the course of the performance tests, the behavior of the staircase join in dependence on the size of the queried XML document was examined and compared to the behavior of the native join algorithm (the nested-loop join) which is chosen by the original PostgreSQL database to evaluate SQL-based XPath expressions. Each of the four test queries consisted of two location steps. The first one, a `descendant` step, provided a reasonable context set for the second one (`descendant`, `ancestor`, `preceding`, and `following`, respectively) which was of actual interest. They were executed in two independent instances of PostgreSQL 7.3.3, an original, unmodified instance and a staircase join-enhanced, tree-aware instance.



(a) Q_{desc} (b) Q_{anc}

(c) Q_{prec} (d) Q_{fol}

Figure 5.2: The execution times of the test queries in both database instances. While nested-loop and staircase join show a very similar behavior with respect to the `descendant` axis (a), the nested-loop join performs considerably inferior in case of the other three test queries.

The performance tests carried out in the tree-aware PostgreSQL instance confirmed that the staircase join leads to a substantial query speed up. In comparison to the original PostgreSQL instance, the improvement amounts up to several orders of magnitude.

As Figure 5.2 illustrates, the evaluation times of the examined XPath axes in the tree-aware database were found to grow linearly to the size of the queried XML documents. In case of the two largest documents F (110 MB) and G (1.1 GB), the staircase join even turned out to be the only reasonable alternative to execute the test queries (except Q_{desc}) at all.

In contrast to the tree-aware database, the execution times in the unmodified instance of PostgreSQL 7.3.3 grew roughly quadratically to the size of the document table. An exception from this observation is the `descendant` axis. Due to shrink-wrapping, an optimization on the SQL level, it showed similar test results as in the tree-aware database. The unmodified PostgreSQL instance uses an index nested-loop join to evaluate the joins in the test queries.

5.3 Outlook

The test results obtained in the tree-aware PostgreSQL instance showed a very promising performance speed-up. However, further improvements could probably be achieved by the implementation of a more flexible index scan as described in the previous sections. Ideally, it should incorporate the execution of partitioning and skipping and allow for the evaluation of index clauses on the basis of two distinct search keys.

Another task important to consider in the future is the development of a cost model that estimates the execution cost of the staircase join in PostgreSQL. It must detect situations in which the staircase join is to be preferred in comparison to other join algorithms and ensure that an optimal join order is chosen to answer a query.

Appendix A

Data Structures in the Planner/Optimizer

This section lists and describes a few data structures which are important for PostgreSQL's planner/optimizer in general and for the implementation of the staircase join in particular. It provides a deeper insight into the processes involved in planning and optimization and prepares the ground for the source code presented in Appendix B.

A.1 The `Query` Structure

The `Query` structure represents the parse tree of a PostgreSQL query after parsing and rewriting. It is the key input to the search for an optimal query execution plan within the planner/optimizer and stores all the information that can be retrieved directly from the PostgreSQL query, such as the type of the query, e.g. `SELECT`, `INSERT`, etc., a list of the relations in the `FROM` clause of the query, a list of result tuple attributes (the target list), the selection clauses, and the join clauses along with references to the joined relations. Apart from that, there are also information about set operations, sub-queries, aggregates, `DISTINCT`, `ORDER BY`, `GROUP BY`, and `HAVING` clauses. The definition of the `Query` structure can be found on the CD included in this thesis under `.../src/include/nodes/parsenodes.h`.

A.2 The `RelOptInfo` Structure

The `RelOptInfo` structure either represents a base relation (a table or the result of a sub-query) or a join relation (the result of a join). The latter stands for a combination of two or more base relations. A join between a certain set of base relations (e.g. R, S, and T) is always represented by the same `RelOptInfo` structure, regardless of the order in which these relations are actually joined. Details on the join order and the join algorithm, i.e. on the join tree, are stored in a list of `Path` structures (`RelOptInfo.pathlist`) which is constantly updated during the dynamic programming algorithm of the PostgreSQL planner/optimizer. If the `RelOptInfo` represents a base relation, this list contains the access paths available for the relation. Note that only sub-optimal results are kept in the list.

Besides, each `RelOptInfo` stores the join clauses in which any of the relations within the `RelOptInfo` are involved and that have not yet been processed (`RelOptInfo.joininfo`). Base relation `RelOptInfo`s additionally keep the selection clauses in which the relation occurs. Both kinds of `RelOptInfo` structures

contain result size and cost estimates. `.../src/include/nodes/relation.h` contains the definition of this data structure.

A.3 The `Path` Structure

A `Path` structure (see `.../src/include/nodes/relation.h`) may either represent an access path to a single relation (sequential or index scan) or a so-called join path. Up to now, PostgreSQL has supported nested-loop, merge, and hash join paths. In the course of our implementation, staircase join paths were additionally incorporated. As the "operands" of a join path are paths, too, this approach results in the creation of join trees in which each parent node stands for a join algorithm and each leaf node for an access path. The final path combines all the joined relations of a query. `Path` structures may also contain information on the sort ordering of its result (`Path.pathkeys`) and specific cost information. They are embedded in `RelOptInfo`s.

A.4 The `PathKeyItem` Structure

Keeping track of sort order is quite important during planning and optimization, because additional sorting steps may cause a significant increase in the cost of an execution plan. For example, the input relations of a merge join must be sorted on the join attributes. Also, result tuples might have to be returned in a predefined order. The sort order of a path is represented by a list of sublists of `PathKeyItem` data structures (`Path.pathkeys`). Each path key item consists of an ordering operator (`PathKeyItem.sortop`) and a representation of the attribute to which the ordering applies (`PathKeyItem.key`). Each sublist of path key items corresponds to one level of sortedness (primary, secondary etc.), i.e., the n-th sublist corresponds to the n-th sort key of the path. The reason for using a list of sublists, instead of a list of single path keys, is that after a merge join, for example, a path may have two (or even more) equivalent sort keys. If, for example, relations R and S are joined using the join condition $R.X = S.Y$ and the outer relation R is sorted on $R.X$, the result of the join will be sorted on both, $R.X$ and $S.Y$. The definition of path key items can be found in `.../src/include/nodes/relation.h`.

A.5 The `RestrictInfo` Structure

`RestrictInfo` structures (see `.../src/include/nodes/relation.h`) are created for selection clauses ($R.X = 3$) and join clauses ($R.X = S.Y$). In case of a join clause, it additionally specifies whether the clause is "mergejoinable" or "hashjoinable" and if so, on the join operator and the required sort order of the joined relations. Our implementation introduced a means to mark a clause as "staircase-joinable".

`RestrictInfo` structures are used in various places. For example, base relations are assigned one such data structure for each selection clause in which the relation is referenced (`RelOptInfo.baserestrictinfo`). Join clauses are also stored as `RestrictInfo`s within the join paths of a join `RelOptInfo` (`RelOptInfo.pathlist`). This is necessary, because different join orders require different join clauses to be applied. For example, the join of relations R, S, and T, which is represented by one and the same `RelOptInfo`, no matter the join order, may be performed by joining $R \bowtie (S \bowtie T)$ or $(R \bowtie S) \bowtie T$. In the first case, the join clause connecting R and S is required as `RestrictInfo`, but in the second case, this predicate was already applied on the previous level and it is the clause relating S and T that is required

here. Furthermore `RestrictInfo` structures are used within the `JoinInfo` data structure (see below).

A.6 The `JoinInfo` Structure

`JoinInfo` structures (see `.../src/include/nodes/relation.h`) consist of a list of relation ids and a list of `RestrictInfo` structures. They are attached to a `RelOptInfo` structure (`RelOptInfo.joininfo`) and specify the relations that remain to be joined to the `RelOptInfo` and the list of join clauses that must be applied in these joins. For example, if the given `RelOptInfo` structure stands for the join between relations R and S and there is the additional join clause $S.X = T.Y$, this clause and the still unjoined relation T are embedded in a join info. If the `RelOptInfo` accommodating R and S had a second additional join info, e.g. $R.V = U.W$ with unjoined relation U, it would be “inherited” to the `RelOptInfo` of R, S, and T.

A.7 The `Plan` Structure

This is the output of the PostgreSQL planner/optimizer. The final plan is derived from the cheapest path detected for the overall query. In this process, each path node is converted into a plan node or so-called subplan, thus a plan also has the form of a tree. The definition of the `Plan` structure can be found in `.../src/include/nodes/plannodes.h`.

Appendix B

New Source Code in the Planner/Optimizer

This section complements the content of Section 3.4. It gives a short description of the most important routines involved in the creation of a staircase join-based execution plan. The routines were either newly introduced into PostgreSQL's planner/optimizer or extended to account for the staircase join as well. The actual source code can be found on the CD which is included into the written version of this thesis (see the path specifications below). Apart from that, the online version of this thesis provides links to an online appendix, which also contains a selection of the most important newly added lines of code.[10] Both documents are located in the KOPS Database of the University of Konstanz (`http://www.ub.uni-konstanz.de/kops/`). The calling hierarchy of the routines is reflected in the indentation of the list entries.

B.1 Clause Preparation

Before the join tree of a query and the associated execution plan can be created, PostgreSQL must determine which join clauses can be used in which join. The following routines identify predicates that qualify for a staircase join.

- `check_strcsjoinable()`

  Located at `.../src/backend/optimizer/plan/initsplan.c`.

  In PostgreSQL, clauses are embedded in `RestrictInfo` data structures. This routine uses the operator and the data types of the clause's operands to determine whether the predicate can be used in a staircase join. If so, the corresponding data fields of the `RestrictInfo` are set accordingly.

  - `op_strcsjoinable()`

    Located at `.../src/backend/utils/cache/lsyscache.c`.

    The actual classification of a clause as "staircasejoinable" takes place on system catalog level. It is based on the system catalog entry associated to the operator of the clause.

    PostgreSQL stores a system catalog entry for each operator. It can be retrieved using the operator's Oid and contains information about the string representation of the operator, its kind (unary or binary), the data

[10] To keep things simple, code fragments that do not refer to the staircase join and/or are nonrelevant for the understanding of the code are left out in the online appendix. Such locations are marked by three vertical dots.

types of its operands, its commutator and negator, etc. The operators allowed in a staircase join furthermore contain information about the sort order required of their operands. If the information stored in the clause comply with the specifications of its operator's system catalog entry, the clause can be used in a staircase join.

B.2 Dynamic Programming

During dynamic programming, PostgreSQL incrementally creates the query's join tree. This process is based on `RelOptInfo`s (see Appendix A.2), which represent both, the input relations and the output of the join.

- `add_paths_to_joinrel()`

  Located at `.../src/backend/optimizer/path/joinpath.c`.

  This routine is called for each possible combination of input `RelOptInfo`s that can be formed during dynamic programming. It considers each available algorithm (staircase, merge, hash, and nested-loop) for the currently planned join.

  - `strcs_inner_and_outer()`

    Located at `.../src/backend/optimizer/path/joinpath.c`.

    If the pre and the post clause and the represented XPath axis can be successfully identified, the creation of a staircase join node is initiated. If the newly created join node offers an advantage in terms of execution cost or sort order, it is stored as `Path` within the `RelOptInfo` which represents the join's output.

    - `get_strcs_type()`
      Located at `.../src/backend/optimizer/path/joinpath.c`.
      This routine identifies the pre and the post clause of the staircase join and, if successful, returns the XPath axis represented by the clauses.
    - `make_pathkeys_for_strcsclauses()`
      Located at `.../src/backend/optimizer/path/pathkeys.c`.
      This routine builds and returns the `PathKeyItem`s that specify the sort order required of the outer and/or inner relation of the staircase join. The computation of the path keys is based on the pre clause.
      - `cache_strcsclause_pathkeys()`
        Located at `.../src/backend/optimizer/path/pathkeys.c`.
    - `create_strcsjoin_path()`
      Located at `.../src/backend/optimizer/util/pathnode.c`.
      This routine creates the staircase join `Path` and estimates its execution cost (always 0 for the moment).

B.3 Conversion into an Execution Plan

If the staircase join path is found to be the cheapest join method for a particular join, it is incorporated into the final execution plan. When a query's join tree is converted into an execution plan, each path node is converted into a plan node.

- `create_strcsjoin_plan()`

  Located at `.../src/backend/optimizer/plan/createplan.c`.

This routine is responsible for the conversion of a staircase join path into a staircase join plan. Apart from that, it prepares the join clauses for the evaluation mechanisms of the executor and inserts explicit sort nodes, if necessary.

Appendix C

Data Structures in the Executor

The following data structures are essential parts of PostgreSQL's executor. They play a major role in the routines presented in Appendix D.

C.1 The `TupleTableSlot` Structure

The executor stores pointers to tuples in a tuple table which consists of `TupleTable-Slots`. Among others, they contain information on the layout of the tuple, i.e. a schema description, and, of course, the actual tuple data. The definition of this data structure can be found in `.../src/include/executor/tupletable.h` on the CD included in this thesis.

C.2 The `EState` Structure

The execution state (see `.../src/include/nodes/execnodes.h`) is the same for all subplans in a query's execution plan. It contains general execution information such as the overall scan direction (forwards or backwards) and a pointer to the result tuple table.

C.3 The `JoinState` Structure

Join states store all the information required for the execution of a particular join. Among others, they contain pointers to various tuple table slots (e.g. for the result tuple, the inner and outer tuple etc.) and the join's expression context (see below). The definition of join states can be found in `.../src/include/nodes/execnodes.h`.

C.4 The `ExprContext` Structure

Expression contexts are used to retrieve the values required to evaluate an expression, including join and index clauses. They contain two tuple table slots in which the currently processed outer and inner tuple are stored for evaluation purposes. When, for example, a join clause is evaluated, its two operands are examined to determine whether they refer to the `OUTER` or `INNER` relation of the join. If the operand is found to address the `OUTER` relation, the required attribute value is retrieved from

the outer tuple table slot. Otherwise, it is retrieved from the inner tuple table slot. Expression contexts are defined in `.../src/include/nodes/execnodes.h`.

Appendix D

New Source Code in the Executor

This section complements the content of Section 3.5. It contains a selection of the most important routines that were added to PostgreSQL's executor during the implementation of the staircase join. The actual code can be found on the CD included into this thesis or in the online appendix at `http://www.ub.uni-konstanz.de/kops/` (KOPS Database of the University of Konstanz). Again, the calling hierarchy of the routines is reflected in the indentation of the list entries.

D.1 Initialization of Staircase Join Execution

Before a staircase join plan can be executed, the join state, the expression context, and the tuple table slots (see Appendix C) must be initialized.

- `ExecInitStrcsJoin()`

  Located at `.../src/backend/executor/nodeStrcsjoin.c`.

  - `SCFormSkipQuals()`

    Located at `.../src/backend/executor/nodeStrcsjoin.c`.
    This procedure prepares additional clauses that will be needed during join execution to carry outer further comparisons (e.g. to evaluate the second partition boundary in case of the `descendant` axis or to apply pruning in case of the `following` axis). The additional predicates will be created on the basis of the pre and the post clause.

D.2 The Staircase Join's Execution Module

The staircase join's execution module was developed as a state automaton. It encapsulates implementations of pruning, partitioning, and skipping and adapts the original staircase join concepts to the pipelining mechanisms of PostgreSQL.

- `ExecStrcsJoin()`

  Located at `.../src/backend/executor/nodeStrcsjoin.c`.

  This is the top-level routine of staircase join execution. It calls the state automatons in dependence on the XPath axis (`descendant`, `ancestor`, `following`, or `preceding`) that is to be evaluated.

- `ExecDescJoin()`

  Located at `.../src/backend/executor/nodeStrcsjoin.c`.

  The state automaton responsible for the evaluation of the `descendant` axis.

- `ExecAncJoin()`

  Located at `.../src/backend/executor/nodeStrcsjoin.c`.

  The state automaton responsible for the evaluation of the `ancestor` axis.

- `ExecFolJoin()`

  Located at `.../src/backend/executor/nodeStrcsjoin.c`.

  The state automaton responsible for the evaluation of the `following` axis.

- `ExecPrecJoin()`

  Located at `.../src/backend/executor/nodeStrcsjoin.c`.

  The state automaton responsible for the evaluation of the `preceding` axis.

D.3 Completing Staircase Join Execution

To finish execution, the staircase join plan must be cleaned up.

- `ExecEndStrcsJoin()`

  Located at `.../src/backend/executor/nodeStrcsjoin.c`.

Appendix E

DTD of the XML Test Documents

The following list specifies the DTD of the XML documents created using the *XML-gen* generator of the *XMark benchmark project* [SWK+02, XMA03].

```
<!-- DTD for auction database -->
<!-- $Id: auction.dtd,v 1.15 2001/01/29 21:42:35 albrecht Exp $ -->

<!ELEMENT site            (regions, categories, catgraph, people,
                           open_auctions, closed_auctions)>

<!ELEMENT categories      (category+)>
<!ELEMENT category        (name, description)>
<!ATTLIST category        id ID #REQUIRED>
<!ELEMENT name            (#PCDATA)>
<!ELEMENT description     (text | parlist)>
<!ELEMENT text            (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold            (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword         (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph            (#PCDATA | bold | keyword | emph)*>
<!ELEMENT parlist         (listitem)*>
<!ELEMENT listitem        (text | parlist)*>

<!ELEMENT catgraph        (edge*)>
<!ELEMENT edge            EMPTY>
<!ATTLIST edge            from IDREF #REQUIRED to IDREF #REQUIRED>

<!ELEMENT regions         (africa, asia, australia, europe,
                           namerica, samerica)>
<!ELEMENT africa          (item*)>
<!ELEMENT asia            (item*)>
<!ELEMENT australia       (item*)>
<!ELEMENT namerica        (item*)>
<!ELEMENT samerica        (item*)>
<!ELEMENT europe          (item*)>
<!ELEMENT item            (location, quantity, name, payment, description,
                           shipping, incategory+, mailbox)>
<!ATTLIST item            id ID #REQUIRED
                          featured CDATA #IMPLIED>
```

```
<!ELEMENT location       (#PCDATA)>
<!ELEMENT quantity       (#PCDATA)>
<!ELEMENT payment        (#PCDATA)>
<!ELEMENT shipping       (#PCDATA)>
<!ELEMENT reserve        (#PCDATA)>
<!ELEMENT incategory     EMPTY>
<!ATTLIST incategory     category IDREF #REQUIRED>
<!ELEMENT mailbox        (mail*)>
<!ELEMENT mail           (from, to, date, text)>
<!ELEMENT from           (#PCDATA)>
<!ELEMENT to             (#PCDATA)>
<!ELEMENT date           (#PCDATA)>
<!ELEMENT itemref        EMPTY>
<!ATTLIST itemref        item IDREF #REQUIRED>
<!ELEMENT personref      EMPTY>
<!ATTLIST personref      person IDREF #REQUIRED>

<!ELEMENT people         (person*)>
<!ELEMENT person         (name, emailaddress, phone?, address?,
                          homepage?, creditcard?, profile?, watches?)>
<!ATTLIST person         id ID #REQUIRED>
<!ELEMENT emailaddress   (#PCDATA)>
<!ELEMENT phone          (#PCDATA)>
<!ELEMENT address        (street, city, country, province?, zipcode)>
<!ELEMENT street         (#PCDATA)>
<!ELEMENT city           (#PCDATA)>
<!ELEMENT province       (#PCDATA)>
<!ELEMENT zipcode        (#PCDATA)>
<!ELEMENT country        (#PCDATA)>
<!ELEMENT homepage       (#PCDATA)>
<!ELEMENT creditcard     (#PCDATA)>
<!ELEMENT profile        (interest*, education?, gender?, business, age?)>
<!ATTLIST profile        income CDATA #IMPLIED>
<!ELEMENT interest       EMPTY>
<!ATTLIST interest       category IDREF #REQUIRED>
<!ELEMENT education      (#PCDATA)>
<!ELEMENT income         (#PCDATA)>
<!ELEMENT gender         (#PCDATA)>
<!ELEMENT business       (#PCDATA)>
<!ELEMENT age            (#PCDATA)>
<!ELEMENT watches        (watch*)>
<!ELEMENT watch          EMPTY>
<!ATTLIST watch          open_auction IDREF #REQUIRED>

<!ELEMENT open_auctions  (open_auction*)>
<!ELEMENT open_auction   (initial, reserve?, bidder*, current, privacy?,
                           itemref, seller, annotation, quantity, type,
                           interval)>
<!ATTLIST open_auction   id ID #REQUIRED>
<!ELEMENT privacy        (#PCDATA)>
<!ELEMENT initial        (#PCDATA)>
<!ELEMENT bidder         (date, time, personref, increase)>
<!ELEMENT seller         EMPTY>
<!ATTLIST seller         person IDREF #REQUIRED>
```

```
<!ELEMENT current         (#PCDATA)>
<!ELEMENT increase        (#PCDATA)>
<!ELEMENT type            (#PCDATA)>
<!ELEMENT interval        (start, end)>
<!ELEMENT start           (#PCDATA)>
<!ELEMENT end             (#PCDATA)>
<!ELEMENT time            (#PCDATA)>
<!ELEMENT status          (#PCDATA)>
<!ELEMENT amount          (#PCDATA)>

<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction  (seller, buyer, itemref, price, date,
                            quantity, type, annotation?)>
<!ELEMENT buyer           EMPTY>
<!ATTLIST buyer           person IDREF #REQUIRED>
<!ELEMENT price           (#PCDATA)>
<!ELEMENT annotation      (author, description?, happiness)>

<!ELEMENT author          EMPTY>
<!ATTLIST author          person IDREF #REQUIRED>
<!ELEMENT happiness       (#PCDATA)>
```

Bibliography

[BBC$^+$03] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. W3C Working Draft, World Wide Web Consortium, http://www.w3.org/TR/xpath20, November 2003.

[BCF$^+$03] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Working Draft, World Wide Web Consortium, http://www.w3.org/TR/xquery, November 2003.

[Bon02] Peter A. Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. Ph.D. thesis, University of Amsterdam, The Netherlands, 2002.

[Gru02] 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.

[GvK03] Torsten Grust and Maurice van Keulen. Tree Awareness for Relational DBMS Kernels: Staircase Join. In H. Blanken, T. Grabs, H.-J. Schek, R. Schenkel, and G. Weikum, editors, *Intelligent Search on XML Data*. Springer Verlag, September 2003.

[GvKT03] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational RDBMS 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.

[GvKT04] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath Evaluation in Any RDBMS. In *ACM Transactions on Database Systems (TODS)*. Association for Computing Machinery, March 2004. (To appear).

[Pos02] The PostgreSQL Global Development Group. *PostgreSQL 7.3.2 Administrator's Guide*, ©1996–2002.

[Sim98] Stefan Simkovics. *Enhancement of the ANSI SQL Implementation of PostgreSQL*. Master's thesis, Technische Universität Wien, November 1998.

[SWK$^+$02] 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 Int'l Conference on Very Large Databases (VLDB)*, pages 974–985. Hong Kong, China, August 2002.

[WD02] John C. Worsley and Joshua D. Drake. *Practical PostgreSQL*. O'Reilly & Associates, Inc., Sebastopol, CA, January 2002.

[XMA03] XMark - An XML Benchmark Project. www.xml-benchmark.org, November 2003.