Relational approach to logical query optimization of XPath

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

ABSTRACT

To be able to handle the ever growing volumes of XML documents, effective and efficient data management solutions are needed. Managing XML data in a relational DBMS has great potential. Recently, effective relational storage schemes and index structures have been proposed as well as special-purpose join operators to speed up querying of XML data using XPath/XQuery. In this paper, we address the topic of query plan construction and logical query optimization. The claim of this paper is that standard relational algebra extended with special-purpose join operators suffices for logical query optimization. We focus on the XPath accelerator storage scheme and associated staircase join operators, but the approach can be generalized easily.

General Terms

XML, relational algebra, query optimization

1. INTRODUCTION

With the gaining popularity of XML, ever growing volumes of XML-documents need to be managed. Existing relational database technology has great potential for being able to manage these volumes of data. Much research is currently directed at relational storage schemes and efficient evaluation of XPath [2] and XQuery [3] inside an RDBMS reusing the heritage of decades of relational technology.

There are effectively two classes of approaches to storage schemes for XML. First, *shredding*-based approaches which store XML data in many relations based on tag names of elements. For example, data on elements `<employee>` is stored in a relation called `Employee`. A representative of this approach is [12]. The disadvantage of these approaches is that they only work well for *data-centric* XML, i.e., highly structured XML with schema, a relatively low number of different tag names, and no mixed content nodes. The other class of approaches to storage schemes views an XML document as a tree and is centered around one relation storing data on a per-node basis. Representatives of this class are [8, 4]. These approaches also work for *document-centric* XML, but generally have difficulty managing updates and the huge number of tuples in the one relation.

It has been observed by many (e.g., [4]) that in some cases, performance of queries is far from optimal. Causes can be found in the fact that the RDBMS ordinarily can only deploy generic index structures and join algorithms, and, secondly, sometimes makes bad choices in constructing a query plan.

The many index structures and numbering schemes that have been proposed to speed up XML queries in relational backends, often come with associated special-purpose joins. Some examples: the *multi-predicate merge join* (MPMGJN) [15] avoids row comparisons compared to a standard merge join when using an information retrieval-style inverted index. [1] generalizes this work by proposing two families of structural joins, tree-merge and stack-tree, of which the MPMGJN is a member of the tree-merge family. Li and Moon suggest three path joins to be used in a numbering scheme based on preorder ranks and subtree sizes [8]. The *staircase join* speeds up location steps in the presence of a preorder/postorder numbering scheme, called XPath accelerator [4, 6]. It fully uses the properties of the numbering scheme to minimize comparisons and avoids sorting and duplicate removal operations [5]. It has been shown that the join can easily be integrated in an ordinary RDBMS [9]. Experiments have proven that the performance benefit can be significant if these special-purpose joins can be effectively utilized.

This brings us to the second cause of suboptimal performance, namely bad query plan construction, which has not received as much attention yet, but is gaining in attention quickly. For example, [14] investigates choosing join order for structural joins of the stack-tree family. The choice is based on the cost model of [13]. In the context of main-memory RDBMSs, [11] investigates a cost model for the XPath accelerator.

This paper focuses on query plan construction and logical query optimization of XPath queries. One can do logical query optimization on XPath level (e.g., using the equivalences of [10]) or in other XPath-specific manners such as [7], but it is our aim to avoid XML-specific techniques and exploit as much existing relational techniques as possible. Although index structures, numbering schemes, and

special-purpose joins are XML-specific, we claim that standard relational algebra extended with these special-purpose join operators suffices for effectively describing query plans and optimizing them.

The observation underlying the work is that many of the proposed joins are only valid or efficient in certain situations, but that choices can often be made on a logical level, i.e., without a cost model. Furthermore, [11] showed that for certain axes, it is hard to estimate intermediate result sizes rendering a cost model sometimes rather inexact. Therefore, we envision an approach of generating an initial query plan, then rewriting it according to equivalence rules using a strategy that focusses on selection of efficient join operators first. Final judgement should obviously be left to a cost estimation. This paper, however, addresses the first two steps.

More concretely, we define a standard relational algebra and extend it with several join operators. The paper specifically uses the XPath accelerator [4, 6] with variants of the staircase join [5]. The approach, however, is generic enough to be adaptable to include more join variants or even to using another relational storage scheme with other join operators. [5] proposed only one staircase join operator. We will show that this is a good choice for one class of XPath expressions (i.e., path expressions without predicates). For a larger class, other variants of the staircase join are beneficial. For this larger class, we will show how an initial query plan can be generated, and subsequently, how it can be rewritten into more efficient plans.

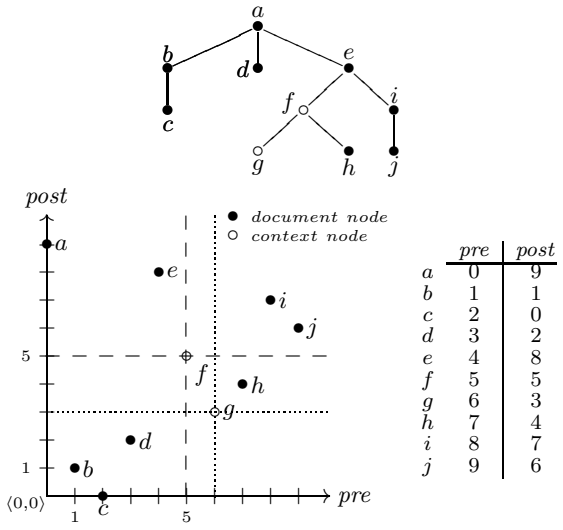
This work is part of a larger effort under the project name *Pathfinder*[1] to construct an XQuery engine on top of relational backends. Among the aims are efficiency, bulk-oriented query processing, and strict adherence to XQuery semantics.

The paper is structured as follows. Section 2 gives a short introduction to the XPath accelerator. Section 3 then describes classes of XPath expressions and defines several staircase join variants beneficial in these classes. Section 4 defines a relational algebra extended with these variants. A typing relation is defined to be able to precisely specify in Section 5, conditions under which certain equivalences hold. Section 6 defines how to generate an initial query plan and Section 7 describes how the equivalence rules can be used to optimize it.

2. XPATH ACCELERATOR

The XPath accelerator [4, 6] is based on a numbering scheme where each node v is assigned a preorder $pre(v)$ and a postorder rank $post(v)$. As many have noted, these numbers efficiently characterize location steps as document regions, also called query windows, e.g., v' is a descendant of $v \Leftrightarrow pre(v) < pre(v') \wedge post(v') < post(v)$. Other properties of the numbering scheme can be more easily seen, when the nodes are placed in a *pre/post plane* according to their preorder (x-axis) and postorder (y-axis) ranks (see Figure 1). XPath expressions can be translated to SQL-queries by simply using the query window associated with the axis step in the WHERE-clause (see Figure 2).

[1] see `http://www.inf.uni-konstanz.de/dbis/research/pathfinder/`



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

Figure 1: *Pre/post* plane and node encoding table `doc`. Lines indicate document regions as seen from context nodes f ("$--$") and g ("..."), respectively.

```
SELECT   DISTINCT v2.*
  FROM   context c, accel v1, accel v2
 WHERE   c.pre ⩽ v1.pre AND v1.post ⩽ c.post
   AND   v1.name = n
   AND   v2.pre < v1.pre AND v2.post < v1.post
   AND   v2.par = v1.par
   AND   v2.kind = text
ORDER BY v2.pre ASC
```

Figure 2: SQL-query for the XPath expression `/descendant-or-self::`n`/preceding-sibling::text()` (*context* is a table containing the root node).

The examples in this paper use the storage scheme of [6] in which each node is represented in a table `doc` with the following attributes: (1) *pre* (preorder rank), (2) *post* (postorder rank), (3) *par* (preorder rank of parent), (4) *kind* (node kind), and (5) *name* (tag name of an element or attribute). Other information, such as the text/value of text and attribute nodes, is stored in other tables (not used in this paper). For example, the tuple $\langle pre = 1, post = 1, par = 0, kind = \text{elem}, name = \text{'b'}\rangle$ describes an element with tag name 'b', which is a child of the root node.

We refer to [6, 4] for more details.

3. STAIRCASE JOIN VARIANTS

Query plans made by an RDBMS for SQL-queries like the one in Figure 2, often use a nested loop join with index scans as operands. Since the RDBMS has no knowledge of properties of pre/postorder ranks, it treats them as ordinary numbers. A special-purpose join, the *staircase join*, is introduced in [5], which uses rank properties to avoid many tuple accesses, duplicate production, order changes, hence, avoids post-processing needed to conform to XPath semantics.

Given the storage scheme of Section 2, the semantics of the staircase join for ancestor is as follows (other major axes analogous):

$$\texttt{context} \underset{\text{anc}}{\overrightarrow{\triangle}} \texttt{doc} =$$
$$\{n \in \texttt{doc} \mid \exists c \in \texttt{context} \bullet c{\cdot}pre > n{\cdot}pre \wedge c{\cdot}post < n{\cdot}post\}$$

In words, it retrieves all nodes from the `doc` table which are an ancestor of some node in `context`. Since it only produces tuples from its right operand, we call it the *right staircase join* to distinguish it from other variants introduced below.

The right staircase join is meant for a specific class of XPath expressions: *path expressions without predicates*, more exactly, path expressions of the form $s_1/s_2/\cdots/s_n$ where $s_i = \alpha :: \tau$ is an axis step along axis α and node test τ. The node sequence output for step s_i is used as context node sequence for step s_{i+1}. An ordinary join would concatenate matching tuples c and n, but attributes of c are irrelevant for step s_{i+1}, hence are discarded.

The next larger class of XPath expressions includes path expressions in predicates, i.e., a step s_i is of the form $\alpha :: \tau[p]$ where p is a *relative* path expression $./s'_1/s'_2/\cdots/s'_m$. An example of an expression of this class is $/s_1[./s_2]$ where $s_1 =$ descendant::a and $s_2 =$ ancestor::b. The output of a join for s_2 between the result of s_1 and `doc` should output 'a' nodes, i.e., context nodes of s_2. This gives rise to the conception of a different staircase join variant, the *left staircase join*:

$$\texttt{context} \underset{\text{anc}}{\overleftarrow{\triangle}} \texttt{doc} =$$
$$\{c \in \texttt{context} \mid \exists n \in \texttt{doc} \bullet c{\cdot}pre > n{\cdot}pre \wedge c{\cdot}post < n{\cdot}post\}$$

In more complex expressions, such as $s_1[./s_2/s_3]$, the left and right staircase join do not suffice, because it is desirable to have a join that outputs information of both joined tuples like a typical join. We, therefore, also introduce the *generalized staircase join* ($a +\!+ b$ is record concatenation):

$$\texttt{context} \underset{\text{anc}}{\triangle} \texttt{doc} =$$
$$\{c +\!+ n \mid \exists c \in \texttt{context}\, \exists n \in \texttt{doc} \bullet c{\cdot}pre > n{\cdot}pre \wedge c{\cdot}post < n{\cdot}post\}$$

From an algorithmic point of view, $\overleftarrow{\triangle}$ and $\overrightarrow{\triangle}$ can apply more optimizations, hence are more efficient, than $\triangle$. Staircase joins are similar to merge joins. They sequentially scan both tables simultaneously while skipping parts of the document table that are guaranteed not to match context tuples. In case of $\overleftarrow{\triangle}$, it suffices to find just one ancestor node for each context node to output the tuple. Therefore, compared to $\overrightarrow{\triangle}$, larger portions of the document table can be skipped. Hence, $\overleftarrow{\triangle}$ is probably the most efficient of the three. $\overleftarrow{\triangle}$ and $\overrightarrow{\triangle}$ can be seen as special cases of $\triangle$ where some of the *attributes are projected out* and resulting duplicates are not generated. It is exactly this observation that drives the logical optimization presented in the sequel.

There are larger classes of XPath expressions that allow more features in the predicates besides relative path expressions. For example, allowing path expressions originating from the root or comparisons with constants or other path expressions. A particularly interesting class is one where functions like *position* and *last* play a role. Preliminary examinations have revealed that algorithms for staircase join variants producing a position attribute are possible, but they are expected to be less efficient than the ones described above. The three given staircase join variants can be seen as special cases of these where also the position attribute is projected out. Consequently, the approach taken here of projection driven join selection will most certainly work for this larger class of XPath expressions as well.

For simplicity of presentation, we concentrate in the sequel on XPath expressions with predicates containing only relative path expressions. The approach is generic enough to be adaptable to the larger classes.

4. EXTENDING RELATIONAL ALGEBRA

In this section, we define a standard relational algebra and extend it with the aforementioned staircase join variants. We first define some relational concepts and record manipulation functions. We have simplified the presentation by, for example, defining record types solely by their attribute labels, excluding the attribute's domains. We then define the extended relational algebra in terms of the set of possible expressions Expr and an associated typing relation '::'.

Definition 1. Let $\mathcal{L}$ be the set of *attribute labels*. We vary l over $\mathcal{L}$. Let $\mathsf{Rec} \in \mathcal{P}\,\mathcal{L}$ be the set of *record types*. We vary r over Rec using notation $r = \langle l_1, \ldots, l_n \rangle$. A label of a record type is called an *attribute*. A *relation* is defined by its record type, so we do not define it separately. *Record concatenation* '$+\!+$' $\in \mathsf{Rec} \times \mathsf{Rec} \rightsquigarrow \mathsf{Rec}$ is defined by $r_1 +\!+ r_2 = r_1 \cup r_2$ if and only if $r_1 \cap r_2 = \emptyset$. The function $\mathsf{prefix} \in \mathcal{L} \times \mathsf{Rec} \rightarrow \mathsf{Rec}$ *prefixes* the attribute labels of a record type: $\mathsf{prefix}(l', \langle l_1, \ldots, l_n \rangle) = \langle l'{\cdot}l_1, \ldots, l'{\cdot}l_n \rangle$. Its dual $\mathsf{unprefix} \in \mathcal{L} \times \mathsf{Rec} \rightsquigarrow \mathsf{Rec}$ removes prefixes: $\mathsf{unprefix}(l', \langle l'{\cdot}l_1, \ldots, l'{\cdot}l_n \rangle) = \langle l_1, \ldots, l_n \rangle$, which is only defined for record types where all labels are prefixed with l'. □

We continue by defining our algebra as a set of expressions. The kernel of the algebra consists of the standard projection π_r, selection σ_p, and join $\bowtie_p$. To improve readability of the query plans, there is also a *negative projection* operator $\tilde{\pi}_r$, which 'projects out' certain attributes. Moreover, prefixing and unprefixing operators $\underset{r}{\triangleleft}$ and $\underset{r}{\triangleright}$ are added to allow for explicit treatment of attribute renaming. Finally and most importantly, we extend the algebra with the three staircase join variants explained in Section 3.

Definition 2. Let Expr be the set of *expressions* inductively defined as in Figure 3. We vary e over Expr. Although we do not specify what a predicate[2] is, we define $\mathsf{labels}(p)$ to return the labels used in p. For readability, we sometimes write the join operators in a prefix way, e.g., $\underset{\alpha}{\triangle}(e_1, e_2)$.

□

[2]We will not further define a *predicate*, but the intention is that it is a boolean expression using connectives and simple comparisons on attributes. A typical predicate would be $pre \leq ctx{\cdot}pre \wedge post \geq ctx{\cdot}post$.

$$\frac{e \in \mathsf{Expr} \quad r \in \mathsf{Rec}}{\pi_r(e), \tilde{\pi}_r(e) \in \mathsf{Expr}} \qquad \frac{e \in \mathsf{Expr} \quad l \in \mathcal{L}}{\underset{l}{\triangleleft}(e), \underset{l}{\triangleright}(e) \in \mathsf{Expr}}$$

$$\frac{r \in \mathsf{Rec}}{r \in \mathsf{Expr}} \qquad \frac{e \in \mathsf{Expr} \quad \text{`}p \text{ is a predicate}^2\text{'}}{\sigma_p(e) \in \mathsf{Expr}}$$

$$\frac{e_1, e_2 \in \mathsf{Expr} \quad \text{`}p \text{ is a predicate'}}{e_1 \bowtie_p e_2 \in \mathsf{Expr}}$$

$$\frac{e_1, e_2 \in \mathsf{Expr} \quad \alpha \in \{\text{anc}, \text{desc}, \text{prec}, \text{foll}\}}{e_1 \underset{\alpha}{\between} e_2, e_1 \underset{\alpha}{\overleftarrow{\between}} e_2, e_1 \underset{\alpha}{\overrightarrow{\between}} e_2 \in \mathsf{Expr}}$$

Figure 3: Expressions

$$\frac{r \in \mathsf{Rec}}{r :: r} \qquad \frac{e :: r' \quad r \subseteq r' \quad r'' = r' \setminus r}{\tilde{\pi}_r(e) :: r''}$$

$$\frac{e :: r' \quad r \subseteq r'}{\pi_r(e) :: r} \qquad \frac{e :: r \quad \mathsf{labels}(p) \subseteq r}{\sigma_p(e) :: r}$$

$$\frac{e :: r}{\underset{l}{\triangleleft}(e) :: \mathsf{prefix}(l, r)} \qquad \frac{e :: r \quad \forall l' \in r \bullet l' = l{\cdot}l''}{\underset{l}{\triangleright}(e) :: \mathsf{unprefix}(l, r)}$$

$$\frac{e_1 :: r_1 \quad e_2 :: r_2 \quad \mathsf{labels}(p) \subseteq r_1 \cup r_2}{e_1 \bowtie_p e_2 :: r_1 +\!\!+ r_2}$$

$$\frac{\begin{array}{c} e_1 :: r \quad \{ctx{\cdot}pre, ctx{\cdot}post\} \subseteq r \\ e_2 :: r' \quad \{pre, post\} \subseteq r' \end{array}}{e_1 \underset{\alpha}{\between} e_2 :: r +\!\!+ r', e_1 \underset{\alpha}{\overleftarrow{\between}} e_2 :: r, e_1 \underset{\alpha}{\overrightarrow{\between}} e_2 :: r'}$$

where $\alpha \in \{\text{anc}, \text{desc}, \text{prec}, \text{foll}\}$

Figure 4: Typing rules

Although XPath is an algebra on node *sequences*, our orderless set-based algebra suffices to capture it. Note that the XPath 2.0 specification [2] states that the result of a location step should be duplicate-free and sorted in document order. As long as the *pre* attribute is not projected out, it captures document order fully. Combined with the duplicate-free restriction, a set-based algebra is sufficient.

Definition 3. We inductively define a typing relation '::'$\in \mathsf{Expr} \leftrightarrow \mathsf{Rec}$ as in Figure 4. $e :: r$ indicates that e is a valid expression and has type r. □

Example 4. Below, a simple example is shown of an algebra expression that can occur as query plan.

$$\pi_{pre}(\sigma_{name=\mathtt{a}}(\underset{ctx}{\triangleleft}(\mathtt{root}) \underset{\text{desc}}{\between} \mathtt{doc}))$$

In words, the expression prefixes the labels of the `root` relation with *ctx*. The result is joined with the `doc` relation using the (generalized) staircase join. This results in a relation with tuples representing the descendants of the root including attributes of the root itself (*ctx*$*$). The expression then selects all tuples with *name* 'a'. Since we are only interested in identifiers of the resulting nodes, the expression projects on the *pre* attribute obtaining the preorder ranks of the descendant nodes with tag name 'a'. □

$$\frac{e :: r}{\pi_r(e) \equiv e} \qquad \frac{e :: r}{\tilde{\pi}_\emptyset(e) \equiv e} \qquad \frac{\pi_r(\pi_{r'}(e)) :: r}{\pi_r(\pi_{r'}(e)) \equiv \pi_r(e)}$$

$$\frac{\pi_r(\tilde{\pi}_{r'}(e)) :: r}{\pi_r(\tilde{\pi}_{r'}(e)) \equiv \pi_r(e)} \qquad \frac{\tilde{\pi}_r(\pi_{r'}(e)) :: r' \setminus r}{\tilde{\pi}_r(\pi_{r'}(e)) \equiv \pi_{r' \setminus r}(e)}$$

$$\frac{\pi_r(\underset{l}{\triangleleft}(e)) :: r}{\pi_r(\underset{l}{\triangleleft}(e)) \equiv \underset{l}{\triangleleft}(\pi_{\mathsf{unprefix}(l,r)}(e))} \qquad \frac{\pi_r(\underset{l}{\triangleright}(e)) :: r}{\pi_r(\underset{l}{\triangleright}(e)) \equiv \underset{l}{\triangleright}(\pi_{\mathsf{prefix}(l,r)}(e))}$$

$$\frac{\pi_r(\sigma_p(e)) :: r \quad Q = \mathsf{labels}(p)}{\pi_r(\sigma_p(e)) \equiv \tilde{\pi}_{Q \setminus r}(\sigma_p(\pi_{Q \cup r}(e)))}$$

$$\frac{e_1 :: r_1 \quad e_2 :: r_2 \quad Q = \mathsf{labels}(p) \quad Q \subseteq r_1 \cup r_2}{\pi_r(e_1 \bowtie_p e_2) \equiv \tilde{\pi}_{Q \setminus r}(\pi_{Q \cup (r \cap r_1)}(e_1) \bowtie_p \pi_{Q \cup (r \cap r_2)}(e_2)}$$

$$\frac{\sigma_p(e_1 \underset{\alpha}{\between} e_2) :: r \quad e_1 :: r_1 \quad \mathsf{labels}(p) \subseteq r_1}{\sigma_p(e_1 \underset{\alpha}{\between} e_2) \equiv \sigma_p(e_1) \underset{\alpha}{\between} e_2}$$

$$\frac{\sigma_p(e_1 \underset{\alpha}{\between} e_2) :: r \quad e_2 :: r_2 \quad \mathsf{labels}(p) \subseteq r_2}{\sigma_p(e_1 \underset{\alpha}{\between} e_2) \equiv e_1 \underset{\alpha}{\between} \sigma_p(e_2)}$$

$$\frac{\sigma_p(e_1 \underset{\alpha}{\overleftarrow{\between}} e_2) :: r}{\sigma_p(e_1 \underset{\alpha}{\overleftarrow{\between}} e_2) \equiv \sigma_p(e_1) \underset{\alpha}{\overleftarrow{\between}} e_2}$$

$$\frac{\sigma_p(e_1 \underset{\alpha}{\overrightarrow{\between}} e_2) :: r}{\sigma_p(e_1 \underset{\alpha}{\overrightarrow{\between}} e_2) \equiv e_1 \underset{\alpha}{\overrightarrow{\between}} \sigma_p(e_2)}$$

$$\frac{\pi_r(e_1 \underset{\alpha}{\between} e_2) :: r \quad e_1 :: r_1 \quad r \subseteq r_1 \quad L = \{ctx{\cdot}pre, ctx{\cdot}post\}}{\pi_r(e_1 \underset{\alpha}{\between} e_2) \equiv \tilde{\pi}_{L \setminus r}(\pi_{L \cup r}(e_1) \underset{\alpha}{\overleftarrow{\between}} e_2)}$$

$$\frac{\pi_r(e_1 \underset{\alpha}{\between} e_2) :: r \quad e_2 :: r_2 \quad r \subseteq r_2 \quad L = \{pre, post\}}{\pi_r(e_1 \underset{\alpha}{\between} e_2) \equiv \tilde{\pi}_{L \setminus r}(e_1 \underset{\alpha}{\overrightarrow{\between}} \pi_{L \cup r}(e_2))}$$

Figure 5: Some equivalence rules

5. EQUIVALENCE RULES

Definition 5. The most important equivalence rules for our extended relational algebra are given in Figure 5. □

The top five equivalence rules of Figure 5 eliminate projections. The given cases often occur as the result of applying other equivalences. For example, the third equivalence states that $\pi_{ctx{\cdot}pre}(\pi_{ctx{\cdot}*}(e))$ is equivalent with $\pi_{ctx{\cdot}pre}(e)$. The typing rules enforce that $\{ctx{\cdot}pre\} \subseteq \{ctx{\cdot}*\}$ or generically in the equivalence rule that $r \subseteq r'$.

The next four rules specify how a projection can be pushed through the (un)prefixing, selection and join operators. Pushing a projection through an (un)prefixing operator means adapting attribute labeling. Since a selection or join condition may refer to attributes that are projected out, they should be retained in the operand and need to be projected

out from the result. For example, $\pi_{pre}(\sigma_{name=\text{a}}(e))$ is equivalent with $\tilde{\pi}_{name}(\sigma_{name=\text{a}}(\pi_{pre,name}(e)))$.

The final six rules show how selection and projection can be pushed through the staircase join variants. If a selection only refers to attributes of the left or right operand of a staircase join, it can be pushed to that operand. Pushing a projection through the staircase join should in a similar way retain the attributes referred to by the staircase join (i.e., *pre*, *post*, *ctx·pre*, and *ctx·post*). The last two rules are of special importance. Notice how pushing the projection through the generalized staircase join, replaces it with the more efficient left or right staircase join. For example, $\pi_{pre}(e_1 \underset{\text{desc}}{\lrcorner} e_2)$ is equivalent with $\tilde{\pi}_{post}(e_1 \underset{\text{desc}}{\overrightarrow{\lrcorner}} \pi_{pre,post}(e_2))$.

For space reasons, we omit correctness proofs of the equivalence rules. Observe that types of equivalent expressions are the same. A standard (set theoretic) semantics of relational algebra suffices for a complete proof. This completes our definition of the extended relational algebra. We now turn our attention to translating XPath expressions to query plans and optimizing them.

6. INITIAL QUERY PLAN CONSTRUCTION

Definition 6. The class of XPath expressions we support, is defined by the grammar below:

xpath ::= '/', *pathexpr*
pathexpr ::= *step* | *step*, '/', *pathexpr*
step ::= *axis*, '::', *nodetest* ['[', *predicate*, ']']
axis ::= self | parent | ancestor | ancestor-or-self
| child | descendant | descendant-or-self
| preceding | preceding-sibling
| following | following-sibling
nodetest ::= node() | element() | text() | *name* | @*name*
predicate ::= './', *pathexpr*

□

Below, we define how to obtain an initial query plan. The function $\mathcal{X}[\![xp]\!](e)$ constructs a plan for (sub)expression *xp*. *xp* may contain a *hole* '$\odot$' to indicate where partial query plan e should be inserted in the resulting query plan.

Definition 7.

[nodetest] $\mathcal{X}[\![\odot :: nodetest]\!](e) = \sigma_p(e)$

nodetest	*p*
node()	*true*
element()	*kind* = elem
text()	*kind* = text
name	*name* = "name"
@*name*	*name* = "name" ∧ *kind* = attr

[axis] $\mathcal{X}[\![\odot/axis]\!](e) = e'$

axis	e'
self	e
child	$\underset{ctx}{\triangleleft}(e) \bowtie_{ctx\cdot pre=parent}$ doc
descendant	$\underset{ctx}{\triangleleft}(e) \underset{\text{desc}}{\lrcorner}$ doc
parent	$\underset{ctx}{\triangleleft}(e) \bowtie_{ctx\cdot parent=pre}$ doc
ancestor	$\underset{ctx}{\triangleleft}(e) \underset{\text{anc}}{\lrcorner}$ doc
preceding	$\underset{ctx}{\triangleleft}(e) \underset{\text{prec}}{\lrcorner}$ doc
preceding-sibling	$\tilde{\pi}_{par_1\cdot *, par_2\cdot *}($ $\sigma_{par_1\cdot parent=par_2\cdot parent}($ $\bowtie_{ctx\cdot pre=par_2\cdot pre}($ $\bowtie_{pre=par_1\cdot pre}($ $\underset{ctx}{\triangleleft}(e) \underset{\text{prec}}{\lrcorner}$ doc $, \underset{par_1}{\triangleleft}(\text{doc}))$ $, \underset{par_2}{\triangleleft}(\text{doc})$ $)))$
following	$\underset{ctx}{\triangleleft}(e) \underset{\text{foll}}{\lrcorner}$ doc
following-sibling	analogous to preceding-sibling

[step₁] $\mathcal{X}[\![\odot/axis :: nodetest[./pathexpr]]\!](e) = \underset{ctx}{\triangleright}(\pi_{ctx\cdot *}(pe))$
where $ax = \mathcal{X}[\![\odot/axis]\!](e)$
$nt = \mathcal{X}[\![\odot :: nodetest]\!](ax)$
$pe = \mathcal{X}[\![\odot/pathexpr]\!](nt)$

This rule is based on the XPath symmetry $p[p_1/p_2] \equiv p[p_1[p_2]]$ for any path expressions p, p_1, and p_2 [10].

[step₂] $\mathcal{X}[\![\odot/axis :: nodetest]\!](e) = nt$
where $ax = \mathcal{X}[\![\odot/axis]\!](e)$
$nt = \mathcal{X}[\![\odot :: nodetest]\!](ax)$

[pathexpr] $\mathcal{X}[\![\odot/step/pathexpr]\!](e) = \mathcal{X}[\![\odot/pathexpr]\!](\mathcal{X}[\![\odot/step]\!](e))$

[xpath] $\mathcal{X}[\![/pathexpr]\!]() = \pi_{pre}(\mathcal{X}[\![\odot/pathexpr]\!](root))$
where *root* is an algebra expression returning a relation with one tuple: the root of the document.

□

Example 8. Example 4 is actually the initial query plan for the simple XPath expression /descendent :: a. It is obtained through the translation scheme above in the following way.

$$\begin{aligned}
&\mathcal{X}[\![/\text{descendent} :: \text{a}]\!]() = \\
&= \pi_{pre}(\mathcal{X}[\![\odot/\text{descendent} :: \text{a}]\!](root)) \\
&= \pi_{pre}(\mathcal{X}[\![\odot :: \text{a}]\!](\mathcal{X}[\![\odot/\text{descendent}]\!](root))) \\
&= \pi_{pre}(\sigma_{name=\text{a}}(\underset{ctx}{\triangleleft}(root) \underset{\text{desc}}{\lrcorner} \text{doc}))
\end{aligned}$$

□

7. QUERY PLAN OPTIMIZATION

We show how the generated plans can be logically optimized by looking at two examples, a simple and a more complex

one. For the latter, two candidate plans are given for which only a cost model-based decision can determine which one is to be preferred.

Example 9. Using the equivalence rules of Figure 5, the initial query plan of /descendent :: a (see Example 8) can be optimized as follows.

$$\mathcal{X}[\![/\text{descendent} :: \text{a}]\!]() =$$
$$= \quad \pi_{pre}(\sigma_{name=\text{a}}(\underset{ctx}{\lhd}(root) \underset{\text{desc}}{\lrcorner} \text{doc}))$$

Push the projection through the selection

$$\equiv \quad \tilde{\pi}_{name}(\sigma_{name=\text{a}}(\pi_{pre,name}(\underset{ctx}{\lhd}(root) \underset{\text{desc}}{\lrcorner} \text{doc})))$$

Push the projection through the staircase join. This selects the more efficient right staircase join variant.

$$\equiv \tilde{\pi}_{name}(\sigma_{name=\text{a}}(\tilde{\pi}_{post}(\underset{ctx}{\lhd}(root) \underset{\text{desc}}{\overrightarrow{\lrcorner}} \pi_{pre,post,name}(\text{doc}))))$$

□

Example 10. For the still considered simple XPath expression /child :: a[./descendant :: b], many semantically equivalent query plans are possible. First, the initial query plan generated by $\mathcal{X}$ (derivation omitted).

$$\pi_{pre}(\underset{ctx}{\rhd}(\pi_{ctx\cdot *}(\sigma_{name=\text{b}}(\underset{\text{desc}}{\lrcorner}(\sigma_{name=\text{a}}(\underset{ctx}{\lhd}(root) \bowtie_p \text{doc}), \text{doc})))))$$
where $p = (ctx{\cdot}pre = par)$

A good strategy for optimization is to push the outer projection inwards as far as possible. The resulting query plan:

$$\underset{ctx}{\rhd}(\tilde{\pi}_{name}(\sigma_{name=\text{b}}(\pi_{ctx\cdot pre,name}(\underset{\text{desc}}{\lrcorner}(\sigma_{name=\text{a}}(\underset{ctx}{\lhd}(root) \bowtie_p \text{doc}), \text{doc})))))$$

Observe that this plan still uses the generalized staircase join, hence is probably rather inefficient. The problem is that the projection $\pi_{ctx\cdot pre,name}(\ldots)$ involves attributes from both operands of the staircase join. In this case, it is probably better to push the selection $\sigma_{name=\text{b}}(\ldots)$ inwards first. This results in the query plan

$$\underset{ctx}{\rhd}(\tilde{\pi}_{ctx\cdot post}(\underset{\text{desc}}{\overleftarrow{\lrcorner}}(\pi_{ctx\cdot pre,ctx\cdot post}(\sigma_{name=\text{a}}(\underset{ctx}{\lhd}(root) \bowtie_p \text{doc})) , \sigma_{name=\text{b}}(\text{doc}))))$$

This query plan can obviously be optimized further by pushing $\pi_{ctx\cdot pre,ctx\cdot post}(\ldots)$ further inwards, but important is that the more efficient left staircase join can be used at the expense of $\sigma_{name=\text{b}}(\ldots)$ being evaluated on a possibly much larger operand. Only a cost model-based evaluation of the query plan can judge whether or not this pays off. □

Note that we only addressed logical optimization on the level of the algebra. Logical optimization can also be done on the level of XPath using the XPath symmetries of [10]. For example, /descendent :: a[./ancestor :: b] is equivalent with /descendent::b/descendent::a. We believe, however, that future research will show that such XML-specific optimization is largely unnecessary, since both expressions will produce equivalent query plans that can probably more effectively be optimized on the level of relational algebra.

8. CONCLUSIONS

In this paper, we claim that for query plan construction and logical optimization for XPath querying on top of an ordinary RDBMS, there is no need for a special algebra, but that a rather standard relational algebra extended with special-purpose XML join operators suffices. The paper shows how, for example, rather standard logical relational optimization techniques can drive join selection for the special-purpose joins. The paper specifically focuses on the XPath accelerator storage scheme and associated staircase join operators to benefit the *Pathfinder* project, but the approach is adaptable to other relational storage schemes and join operators.

The paper includes an extension of a rather standard relational algebra with several staircase join variants. A typing relation has been defined to be able to precisely define the conditions under which certain equivalences hold in the algebra. A translation scheme is given that generates a query plan for a class of XPath expressions that includes predicates with relative path expressions. The equivalence rules are then used to logically optimize the plan. Common strategies are to push projections and selections inwards through the join operators. For several examples, it has been shown how this can drive the application of more efficient join operators. Larger classes of XPath expressions have been sketched for to which the approach can be easily extended. Although this shows that the optimization approach is useful and realistic, full treatment involves, among others, a complete set of equivalence rules and a definition of a rewriting strategy that pursues a normal form.

Other logical next steps in this research are to include cost estimation of query plans, e.g., based on [11], supported by experimental validation. Furthermore, it is interesting to investigate how the rather simple relational equivalence rules compare to the XPath-specific equivalence rules of [10]. We expect that XPath-specific optimization based on these rules is largely unnecessary, because equivalent XPath expressions will map to equivalent relational query plans that can (possibly more effectively) be optimized on the level of the relational algebra. The use of schema information (in a DTD or XML schema) for query optimization is an interesting direction as well. A possible first step here is to investigate how schema information can be used for more accurate cost modelling. Finally, in the context of XQuery, perhaps one or two additional extensions are needed to the otherwise standard relational algebra, but we expect that it is straightforward to construct a few new equivalence rules that include those extensions.

9. REFERENCES

[1] S. Al-Khalifa, H. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of the 18th Int'l Conf. on Data Engineering (ICDE), San Jose,*

California. IEEE Computer Society, Feb. 2002.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. Technical report, World Wide Web Consortium, Nov. 2003. `http://www.w3.org/TR/xpath20/`.

[3] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, Nov. 2003. `http://www.w3.org/TR/xquery`.

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

[5] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of the 29th Int'l Conf. on Very Large Data Bases (VLDB)*, Berlin, Germany, Sept. 2003.

[6] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in Any RDBMS. *ACM Transactions on Database Systems (TODS)*, 29(1):91–131, Mar. 2004.

[7] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized Translation of XPath into Algebraic Expressions Parameterized by Programs Containing Navigational Primitives. In *Proc. of the 3rd Int'l Conf. on Web Information Systems Engineering (WISE), Singapore*. IEEE Computer Society, Dec. 2002.

[8] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 361–370, Rome, Italy, Sept. 2001. Morgan Kaufmann Publishers.

[9] S. Mayer. Enhancing the Tree Awareness of a Relational DBMS: Adding staircase join to PostgreSQL. Master's thesis, University of Konstanz, Germany, 2004.

[10] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Symmetry in XPath. Technical Report PMS-FB-2001-16, Institute of Computer Science, University of Munich, Germany, 2001.

[11] H. Rode. Cost Models for XPath Query Processing in Main Memory Databases. Master's thesis, University of Konstanz, Germany, 2003.

[12] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proc. of 26th Int'l Conf. on Very Large Data Bases (VLDB), Cairo, Egypt*, pages 65–76. Morgan Kaufmann, Sept. 2000.

[13] Y. Wu, J. M. Patel, and H. Jagadish. Estimating Answer Sizes for XML Queries. In *Proc. of the 8th Int'l Conf. on Extending Database Technology (EDBT)*, pages 590–608, Prague, Czech Republic, Mar. 2002.

[14] Y. Wu, J. M. Patel, and H. Jagadish. Structural join order selection for xml query optimization. In *Proc. of the 19th Int'l Conf. on Data Engineering (ICDE), Bangalore, India*. IEEE Computer Society, Mar. 2003.

[15] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 425–436, Santa Barbara, California, May 2001. ACM Press.