

# Relational Approach to XPath Query Optimization

R. Verhage  
Database Group, Faculty of Computer Science  
University of Twente

Supervisors:  
Dr. ir. M. van Keulen  
Dr. ir. D. Hiemstra

July 26, 2005

## **Abstract**

This thesis contributes to the Pathfinder project which aims at creating an XQuery compiler on top of a relational database system. Currently, it is being implemented on top of MonetDB, a main memory database system. For optimization and portability purposes, Pathfinder first compiles an XQuery expression into its own relational algebra, before translating the query into the query language of the underlying system (in this case MIL, the MonetDB Interpreter Language). This thesis focusses on the optimizability aspect of the algebra. Pathfinder's relational algebra is built on a set of basic relational (comparable to SQL) operators extended with the staircase join operator, an operator specifically designed for evaluation of XPath axis steps. By formally specifying the relational model of Pathfinder and all operators of its algebra, this thesis shows how they can be exploited for optimization. The main focus is on the staircase join operator, which, as it seems, does not provide enough rewriting flexibility in its current implementation. Therefore, a new staircase join definition is proposed: the symmetric staircase join.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	XML trees in an RDBMS . . . . .	3
1.1.1	XPath accelerator . . . . .	3
1.1.2	Staircase join . . . . .	5
1.2	MonetDB . . . . .	8
1.2.1	BATs . . . . .	8
1.2.2	Data type void . . . . .	9
1.2.3	MIL . . . . .	9
1.3	Pathfinder . . . . .	10
1.4	Research question . . . . .	11
1.5	Demarcation and deliverables . . . . .	11
1.6	Thesis outline . . . . .	12
<b>2</b>	<b>Related research</b>	<b>13</b>
2.1	Natix . . . . .	13
2.2	Reverse axis removal . . . . .	14
2.3	Other systems . . . . .	14
<b>3</b>	<b>XQuery to algebra compilation</b>	<b>15</b>
3.1	Introducing the algebraic expressions . . . . .	15
3.2	Loop lifting . . . . .	17
3.2.1	Loop lifting with bound variables . . . . .	18
3.2.2	Loop lifting with free variables . . . . .	19
3.3	Loop-lifted staircase join . . . . .	22
3.4	Query plan construction . . . . .	23
<b>4</b>	<b>Pathfinder’s relational algebra</b>	<b>27</b>
4.1	Relational structure . . . . .	27
4.2	Specifying the algebra . . . . .	28
4.3	Rewriting expressions . . . . .	30
<b>5</b>	<b>Staircase join</b>	<b>33</b>
5.1	Staircase join variants . . . . .	33
5.2	Exploring possible equivalence rules . . . . .	35

5.2.1	Commutativity . . . . .	35
5.2.2	Associativity . . . . .	35
5.2.3	Asymmetric staircase join . . . . .	36
5.3	Symmetric staircase join . . . . .	36
5.4	Commutativity and associativity revisited . . . . .	39
5.4.1	Commutativity . . . . .	39
5.4.2	Associativity . . . . .	41
5.5	Pushing operators through a staircase join . . . . .	46
<b>6</b>	<b>Xpath symmetries</b>	<b>47</b>
6.1	Location path equivalence . . . . .	47
6.2	A simple XPath symmetry . . . . .	48
6.2.1	Compiling the equation . . . . .	49
6.2.2	DAG Comparison . . . . .	50
6.3	Pattern recognition . . . . .	50
6.4	Left staircase join . . . . .	50
6.5	Symmetric staircase join . . . . .	52
6.6	Complex XPath expressions . . . . .	53
<b>7</b>	<b>Conclusion</b>	<b>54</b>
7.1	Summary . . . . .	54
7.2	Conclusions . . . . .	56
7.3	Future research . . . . .	57
<b>A</b>	<b>DTD for auction.xml</b>	<b>61</b>
<b>B</b>	<b>XQuery Core dialect</b>	<b>64</b>

# List of Figures

1.1	The four major XPath axes as seen from context node $f$ : (a) preceding, (b) descendant, (c) ancestor and (d) following.	2
1.2	$Pre/post$ plane and the corresponding node encoding table $doc$ for the XML document of figure 1.1. Dashed and dotted lines indicate the document regions as seen from context nodes $f$ and $g$ respectively.	4
1.3	SQL query and associated query plan.	4
1.4	(a) Intersection and inclusion of the <b>ancestor-or-self</b> paths of a context node sequence. (b) The pruned context node sequence covers the same <b>ancestor-or-self</b> region and produces less duplicates (3 rather than 11).	5
1.5	Pruning the <b>descendant</b> axis in the $pre/post$ plane.	6
1.6	Partitioning after pruning in the $pre/post$ plane.	7
1.7	Skipping technique for <b>descendant</b> axis.	7
1.8	Skipping after partitioning in the $pre/post$ plane.	8
1.9	Architecture of the Pathfinder system.	10
3.1	Operators of the relational algebra, $a$ and $b$ being column names.	16
3.2	A typical XQuery <b>for-return</b> construct.	17
3.3	Nesting and different variable scopes.	18
3.4	Algebraic query plan of example 3.3	24
3.5	Algebraic query plan of example 3.4	26
4.1	Atomic values in Pathfinder.	28
4.2	Imposed conditional properties by axis step $\alpha$ .	31
5.1	Truth table for $cond_{x.iter,y.iter}$ with (1) $x.iter = y.iter$ , (2) $x.iter \neq 0 \wedge y.iter = 0$ , (3) $x.iter = 0 \wedge y.iter \neq 0$ and (4) $\neg(x.iter = 0 \wedge y.iter = 0)$ .	39
5.2	Combinations of $iter$ for which equation 5.8 is valid, $n \in \{1, 2, 3, \dots\}$	46
6.1	Symmetrical XPath axis pairs.	48

6.2	DAG corresponding to the left hand side of equation 6.1 . . .	49
6.3	DAG corresponding to the right hand side of equation 6.1 . .	51
6.4	Comparing the two DAGs . . . . .	51
6.5	DAG introducing the <i>left</i> staircase join. . . . .	52
6.6	DAG introducing the <i>symmetric</i> staircase join. . . . .	53

# Chapter 1

## Introduction

Over the past few years, XML (eXtensible Markup Language) has become more and more popular. As a consequence, ever growing volumes of XML documents need to be managed. Existing relational database management systems (RDBMSs) have great potential for being able to manage these volumes of data. Unlike designing a special purpose database system for storing XML from scratch, developing one on top of an existing RDBMS provides great advantages. Relational database development has been going for a few decades, resulting in the efficient RDBMSs of today. Problems such as data consistency, efficient memory management and concurrency have been around for so long that present solutions are very robust and efficient. Added that relational database technology has proven to be able to successfully host types of data which formerly had not been anticipated to live inside an RDBMS (for example spatiotemporal data and complex objects), it is worth the effort to try using relational database technology for efficient handling of XML data.

As XML provides a tree storage scheme for storing data, it also provides its own query languages XQuery and XPath [1], the latter in general being a subset of the former. Both are proposed standards by the World Wide Web Consortium (W3C). In order to create a fully capable XML database management system on top of a relational backend, the requirements are roughly two-fold:

1. Mapping the XML document tree onto the RDBMS' relational table structure;
2. Building an interpreter capable of converting XQuery expressions to the relational algebra (or SQL) used by the RDBMS.

This master's thesis contributes to the Pathfinder project, by looking at possibilities for algebraic query optimization within Pathfinder. Pathfinder is a cooperative project by the Universities of Konstanz (Germany) and

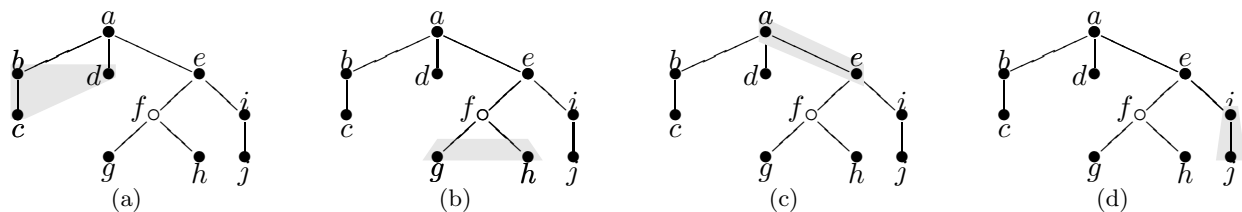


Figure 1.1: The four major XPath axes as seen from context node  $f$ : (a) preceding, (b) descendant, (c) ancestor and (d) following.

Twente (The Netherlands) and CWI (Institute for Mathematics and Computer Science Research) in Amsterdam (The Netherlands).

**Pathfinder** The Pathfinder project aims at creating an XQuery compiler for a number of different relational database backends. The project was initiated by Torsten Grust and Jens Teubner of the Database and Information Systems Group at the University of Konstanz, soon joined by the Database Group of the University of Twente. Later, CWI in Amsterdam joined the project. Currently, Pathfinder is being implemented on top of the main memory database system MonetDB [2] developed at CWI.

As stated before, there are two main goals the project has to target. First, XML documents have to be mapped onto the relational table structure of the RDBMS. In other words: the relational system has to be made *tree aware*. The encoding (mapping) of XML data used in Pathfinder is called the XPath accelerator [3]. In combination with an added join operator, the *staircase join* [4], it provides for a relational structure with full awareness of all 13 XPath axes, the most important being *ancestor*, *descendant*, *preceding* and *following*. Figure 1.1 shows these axes for the sample XML document below.

```

<a>
  <b><c></b>
  <d>
  <e>
    <f><g><h></f>
    <i><j></i>
  </e>
</a>

```

The following section describes both XPath accelerator and the staircase join, the two building blocks used to establish efficient handling of XML trees within an RDBMS. Section 1.2 points out the characteristics of MonetDB relevant to Pathfinder. Putting it all together, section 1.3 explains the architecture of Pathfinder in the implementation on top of MonetDB.

## 1.1 XML trees in an RDBMS

Relational database systems store their data in a simple and straightforward table structure. As mentioned earlier, this structure has proven to be very versatile in the past, by managing data it was not originally designed for. However, XML documents are tree structures, for which an RDBMS in general contains no inner logic able to efficiently handle these structures. In Pathfinder, this logic is provided by the proposed XPath accelerator and staircase join.

### 1.1.1 XPath accelerator

XPath Accelerator maps the XML structure onto relational tables using a numbering scheme based on the *pre-* and *postorder* ranks of all document nodes. A node's preorder rank is determined by traversing the XML document tree starting at the root and numbering all elements in order of occurrence starting at zero. Dual to this preorder traversal is the postorder traversal, which assigns a postorder rank to each node after all its children have been traversed from left to right. In other words, preorder ranks are assigned in order of opening tags, postorder ranks are assigned in order of closing tags. Figure 1.2 shows the assignment of pre- and postorder ranks to the nodes of the XML document depicted in figure 1.1. Note that *pre* order ranks denote the order of the opening tags in the document, as the *post* order ranks denote the order of the corresponding closing tags. This is no coincidence. The figure also shows how the *pre-* and *postorder* ranks can be plotted in the *pre/post* plane. Taking context node *f*, the plot can be partitioned into four regions, characterizing the four major xpath axes. Going clockwise from the upper left region, the four regions represent: *f/ancestor* = (*a, e*), *f/following* = (*i, j*), *f/descendant* = (*g, h*) and *f/preceding* = (*b, c, d*) respectively. This characterization applies to all nodes in the document tree. For example: *g/ancestor* = (*a, e, f*).

Using the encoding of XPath accelerator, XPath expressions can be translated into corresponding SQL queries that will compute the same result. Taking initial context node *f* of figure 1.2 along with the XPath expression

```
following::node()/descendant::node(),
```

the result will be:

$$(f)/following/descendant = (j) .$$

Assuming that the RDBMS has a table *doc* loaded with the *pre/post* node encodings, the path expression above can be translated into the SQL query of figure 1.3. The shown associated query plan is the actual query plan chosen by the optimizer of IBMs DB2 V7.1 [4]. By using two index scans on

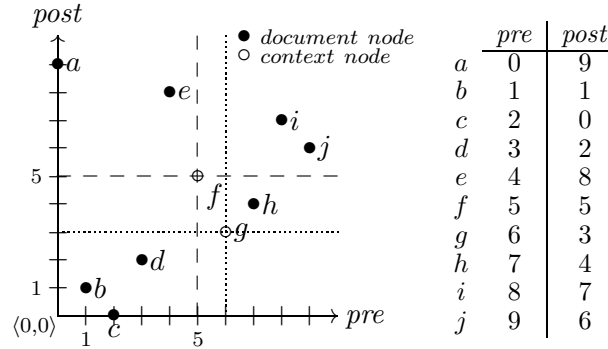


Figure 1.2: *Pre/post* plane and the corresponding node encoding table **doc** for the XML document of figure 1.1. Dashed and dotted lines indicate the document regions as seen from context nodes *f* and *g* respectively.

```

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

```

Figure 1.3: SQL query and associated query plan.

the *pre/post* encodings, the query can be executed fairly efficient. However, to speed-up query processing time, the index range scan might be delimited by adding some more tree knowledge. In any tree, for a node *v*, the size of the subtree below *v* can be estimated by:

$$|(v)/\text{descendant}| = \text{post}(v) - \text{pre}(v) + \underbrace{\text{level}(v)}_{\leq h}$$

Here, *level(v)* denotes the length of the path from the root to *v*, which is bound by *h*, the *height* of the entire document tree. Now, an additional predicate can be added to the SQL query of figure 1.3 (line 7) that delimits the range of the index scan:

```

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

```

Using this subtree size estimation, the number of *pre/post* encodings touched by scanning the **doc** table will be smaller. Still, there will be a significant number of nodes that are touched, but do not occur in the query result. For example, when searching for all *descendants* of a particular node, many of its *preceding* and *following* nodes will also be touched in the process. Since

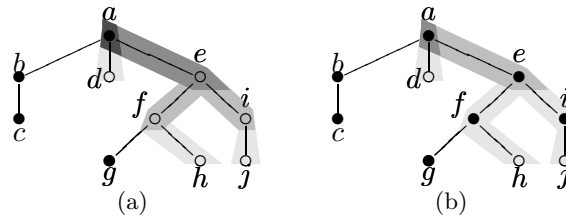


Figure 1.4: (a) Intersection and inclusion of the **ancestor-or-self** paths of a context node sequence. (b) The pruned context node sequence covers the same **ancestor-or-self** region and produces less duplicates (3 rather than 11).

node distribution in the *pre/post* table is not arbitrary, large portions might be skipped. Thus ideally, index scans should be guided to touch only those nodes that constitute the actual query result.

All this tree knowledge for further optimization is available through the *pre/post* encodings, but not to the query optimizer. To do so, the tree knowledge has to be made explicit on the SQL level. To establish this, a new operator is added on the RDBMS' algebraic level: the staircase join.

### 1.1.2 Staircase join

To make the RDBMS tree aware, a new join operator is introduced to its relational algebra, the staircase join. This new operator exploits and encapsulates all tree knowledge that is present in the *pre/post* plane.

By using techniques called *pruning* and *skipping* the staircase join aims at minimizing the number of nodes touched in query evaluation. The next two paragraphs explain the effects of pruning and skipping. A more detailed description of these two techniques along with some basic algorithms can be found in [4]. Throughout these paragraphs it will become clear why this new join operator is called the *staircase* join.

**Pruning** Evaluating XPath expressions, very often there will be a sequence of context nodes on which an axis step is performed. Independent evaluation of this axis step for all nodes in the context may lead to duplication of work. For the **ancestor-or-self** axis, this is illustrated in figure 1.4 for the context node sequence  $(d, e, f, h, i, j)$ . The shaded paths indicate the number of times a node will be produced in the result. For example, node  $a$  will be returned once for every node in the context sequence, while node  $f$  will only be returned twice, once for context node  $h$  and once for itself (figure 1.4 (a)). It is obvious that the number of duplicate nodes in the result can be reduced by shrinking the context node sequence to  $(d, h, j)$  by removing nodes  $(e, f, i)$ , which results in the shaded tree of figure 1.4 (b). The process of eliminating included paths by removing context nodes is called *pruning*.

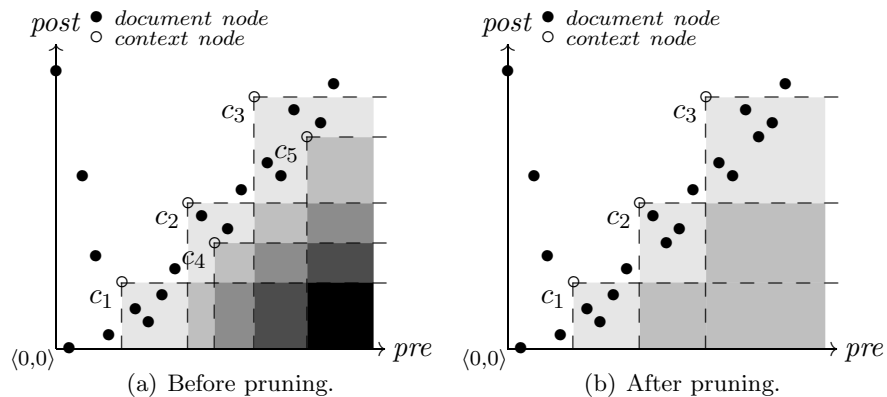


Figure 1.5: Pruning the **descendant** axis in the *pre/post* plane.

The effect of pruning can be more clearly seen in the *pre/post* plane. Figure 1.5 displays the effect of pruning on evaluation of the **descendant** axis for a certain context node sequence  $c_i$ . Nodes  $c_4$  and  $c_5$  in figure 1.5 (a) are eliminated from the context sequence which produces the proper staircase shown in figure 1.5 (b).

**Partitioning** Pruning has already reduced the work load by removing redundant nodes from the initial context set. This has been demonstrated in figure 1.5. However, after pruning, there are still some overlapping regions as can be seen in figure 1.5 (b). These regions are guaranteed to not contain any nodes. The technique used to exclude these regions from the scan over the *pre/post* table is called partitioning. The effect of partitioning is shown in figure 1.6. Since no more regions overlap, partitioning has ensured that one sequential scan over the *pre/post* table is enough to evaluate an XPath axis step.

**Skipping** The effects of pruning and partitioning are obvious and will remove a significant amount of duplicates in the resulting node sequence of an axis step. However, as can be seen in figure 1.5 (b), there might still exist duplicates. Pruning has removed context regions included by others, but does not touch partially overlapping regions. In other words, pruning only affects context nodes in the same direction as the desired axis step. In terms of the **descendant** axis, this means that pruning does not look in the direction of the **following** axis. This is where skipping comes in.

It has already been mentioned that the *pre/post* encoding of XPath accelerator provides some nice properties. Viewed in the *pre/post* plane, for every node in the document tree, four regions can be depicted, corresponding to the four major xpath axes. Looking at a particular context node  $c$ , all its descendant nodes  $d$  will be located in the corresponding lower right region.

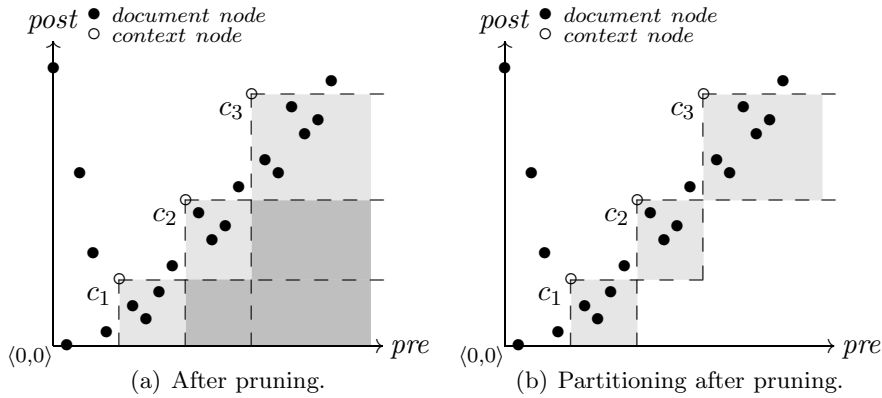


Figure 1.6: Partitioning after pruning in the  $pre/post$  plane.

In other words:  $d.post < c.post$ . All other nodes, not residing in this region, cannot possibly be descendants of  $c$ . As a consequence, when scanning the  $pre/post$  plane from left to right, searching for descendants of node  $c$ , scanning can be terminated with the occurrence of a node  $v$  for which  $v.post > c.post$  holds. Node  $v$  is the first node outside of the context node  $c$ 's descendant region and therefore, no other node beyond  $v$  can possibly be a descending node of  $c$ . In terms of XPath semantics,  $v$  is a *following* node of  $c$ . This means that when such a node  $v$  is encountered, both  $v$  and all nodes between  $v$  and the next context node can be skipped. Figure 1.7 shows the effect of skipping for the descendant axis. Pursuing the descendant example of the previous paragraph, the result of skipping after partitioning is shown in figure 1.8.

Equipped with the techniques of pruning, partitioning and skipping, the staircase join operator can evaluate XPath axis steps in one sequential scan

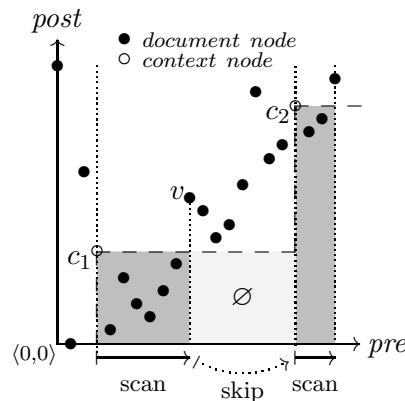


Figure 1.7: Skipping technique for descendant axis.

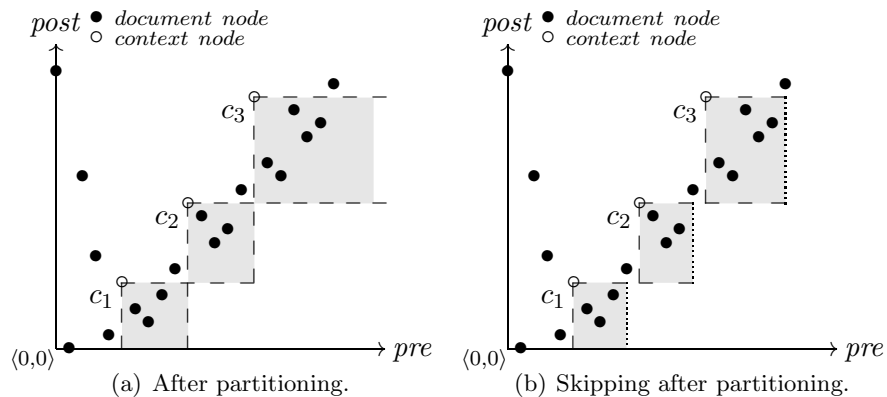


Figure 1.8: Skipping after partitioning in the *pre/post* plane.

over the *pre/post* encoding table. Due to pruning, partitioning and skipping, only those nodes that contribute to the result will be touched. Wrapping up, the staircase join is a very efficient operator that fully exploits XPath accelerator to make the RDBMS tree aware.

## 1.2 MonetDB

As mentioned earlier, Pathfinder is currently being implemented on MonetDB. MonetDB is a main memory database system and one of the first database systems to focus on exploiting CPU caches for query optimization. Along with innovations in this area, MonetDB also deploys new concepts for data storage and table access. There are three features of MonetDB that are important to Pathfinder:

1. Binary Association Tables (BATs);
2. Data type void;
3. Monet Interpreter Language (MIL).

Following three subsections explain the role of these concepts within the MonetDB system.

### 1.2.1 BATs

On the data storage level, MonetDB only supports the use of *Binary Association Tables* (BATs). Every relation in MonetDB is vertically fragmented into BATs. Each BAT consists of two columns, a *head*, often containing a unique identifier and a *tail* for storing attribute data. As a consequence,

to get a full table view of a relation, its BAT fragments have to be joined together <sup>1</sup>.

The choice of using a data storage model with binary tables only is to minimize main memory access, which has become the main bottleneck for database systems running on modern hardware architectures. Improving cache utilization will deal with this problem and this is where BATs come in. Typical database operations only involve a few attributes of a relation. Examples are selection on a single attribute or a simple join comparing only two attributes. With normal relational table storage, whole records are being processed during these operations. For large records, this will mean that a lot of data irrelevant to the query is loaded into the CPU's cache lines. Since vertical fragmentation of data into BATs only stores one attribute per BAT, records will be significantly smaller and more relevant data can be loaded into a single cache line, causing less main memory lookups.

### 1.2.2 Data type void

In terms of data storage size, however, only having BATs will cause major redundancy. Vertical fragmentation requires to maintain a (primary) key in every single table. Apart from storage redundancy, performance will be undermined by the great number of key joins that will occur for complex queries. MonetDB therefore provides a void data type for BAT columns, containing *virtual object identifiers*. Where key values usually form a dense consecutive integer sequence, for a BAT containing a column of type void the system just keeps the offset and length of the key sequence. The BAT can now be handled the same way as a one-dimensional array, greatly reducing data storage. Side benefit is that using simple *offset + key* positional memory lookups, allows for very efficient join algorithms. A disadvantage of the void data type approach is that a void BAT requires its tail to be of a fixed length data type. To overcome this restriction on data type length, variable sized data will be stored on a separate heap, pointed to by fixed length pointers stored in the BAT's tail.

### 1.2.3 MIL

Together with the database backend MonetDB provides its own Monet Interpreter Language (MIL), a rather straightforward interface to query stored data. MIL gives the user a set of algebra operators specifically designed to handle BAT structures and the supported data types. It is a procedural language, providing basic control structures and also providing great extensibility by allowing user defined functions. Within Pathfinder, this extensibility is used to make MonetDB capable of efficient handling of XML

---

<sup>1</sup>For ease of reading, this thesis treats relational tables as having a possible  $n$  columns. Physically, all these tables are still vertically fragmented in BATs.

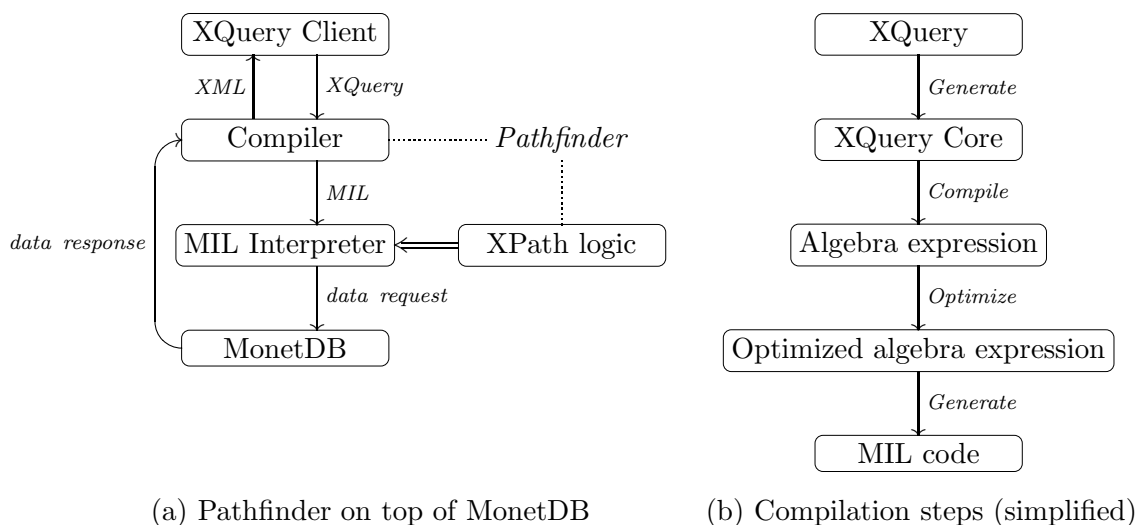


Figure 1.9: Architecture of the Pathfinder system.

trees by implementing the staircase join.

As opposed to other systems, MonetDB makes its internal language available to the user. Most other database management systems act as a black box. They take SQL as their input and produce a result without providing control over the algebra internally used. As opposed to these systems, MIL is directly accessible to the user, making it possible to create an XQuery frontend that can query MonetDB directly. The absence of a top level query language (such as SQL), means that no additional translations need to be made, causing less overhead.

### 1.3 Pathfinder

As stated in the beginning of this chapter, the Pathfinder project aims at creating an XQuery compiler for a number of different relational backends. In this section, the basic architecture of Pathfinder is explained as it is currently being developed on top of MonetDB.

In figure 1.9, a schematic view of the architecture of Pathfinder is given. Figure 1.9 (a) shows how Pathfinder is implemented on top of MonetDB. The compiler takes an XQuery as its input, translates it into the Monet Interpreter Language MIL, which is then passed to MonetDB's MIL interpreter. Along with the added necessary XPath logic, the data can now be fetched from the storage backend and returned to the XQuery compiler. The query result is then translated back to XML and returned to the user.

As stated above, to query the underlying MonetDB database, every XQuery input has to be translated to MIL. This translation is performed

fairly straightforward. However, the resulting MIL code might be far from optimal. More so, the XQuery input might not be optimal as well. In other words, there is still need for some query optimization. Since MIL is a very low level language with just the basic operations needed, the resulting MIL queries will be large (especially when many axis steps are taken) and not easy to read. As a consequence, optimization here is cumbersome and recognizing patterns for optimization might be a near impossible task. Optimizing on the level of the XQuery input might yield some good results, but leave possible optimizations on the relational level unobserved.

To overcome this problem, an intermediate relational algebra is introduced. Each XQuery input will first be compiled into this algebra. To ease compilation, the XQuery is first rewritten into its XQuery Core normal form, see [6], appendix A. The relational algebra is quite primitive, with operators designed to match the capabilities of modern SQL-based systems. A detailed description of the algebra and the compilation procedure can be found in [5]. The resulting algebra expression will then undergo an optimization process before the final translation to MIL code. All these steps are graphically depicted in figure 1.9 (b).

## 1.4 Research question

The previous section points out areas of possible optimization within the Pathfinder system. In terms of the architecture, these areas are *a)* the original XQuery; *b)* its XQuery Core normal form; *c)* the compiled algebraic expression and *d)* the resulting MIL code. For reasons previously explained, best results are expected within the area of algebraic expressions.

This thesis will take a thorough look at these expressions. The research question is: *"In what possible ways could optimization be performed on the algebraic level?"*.

## 1.5 Demarcation and deliverables

Optimization in general is more or less an abstract term and optimization problems tend to be very large. Already, the research question has been narrowed down to the area of compiled algebraic expressions. Within this area, the main focus will be on compiled XPath expressions, involving the staircase join operator. The optimization process will be based on algebraic equivalence rules. By applying these rules, algebra expressions can be rewritten in (possibly) more optimal expressions. Pathfinder has the ability to process queries that address multiple documents. This research will only focus on single document queries.

For testing purposes, there is a Pathfinder prototype written in the Haskell [7] functional programming language. The Pathfinder system itself

is entirely written in C. In C, the algorithms for rewriting and compilation of the algebraic expressions are large and difficult to read. Prototyping in Haskell provides some advantages. The functional programming paradigm is based on evaluating functions (rules) in a non-sequential way. Rules for rewriting and compilation can be almost literally translated into Haskell functions. Where the algorithms written in C will be large and unreadable, the Haskell equivalent will be small and highly readable. As a consequence, the Haskell prototype is much easier to maintain.

To wrap up, the plan of work is to:

1. make a thorough study of Pathfinder's relational algebra;
2. set up a collection of equivalence rules for optimization;
3. implement these rules in the Haskell prototype;
4. experiment with the prototype to assess the impact of the various rules.

Steps 2 to 4 form an iterative prototyping process starting with simple rules for optimization yielding a first prototype. After assessing the impact of these rules and with the experience gained, new rules might be constructed. The process will repeat itself until ideally, no more possible optimization rules might be thought of.

All research as described in this thesis will deliver an optimization module that extends the current Haskell prototype of the Pathfinder system. Furthermore, after completion, a paper will be written on construction of all made up equivalence rules for the relational algebra used within Pathfinder.

## 1.6 Thesis outline

In this chapter, the reader has been introduced to the research material essential for this study. The next chapter gives a short description of research related to the subjects covered in this thesis. Chapter 3 provides an overview of the relational algebra used by Pathfinder and gives an example on how an XQuery expression is compiled into a relational query plan. Chapter 4 explains the aspects of the optimization process by giving a detailed specification of all algebraic operators and showing how these specifications can be used to obtain equivalence rules. Chapters 5 and 6 deal with the most important aspect of XQuery optimization: the optimization of XPath location steps. First, the staircase join operator is discussed in chapter 5. Chapter 6 takes a closer look at XPath symmetries and how they might be recognized in an algebra expression. Finally, the thesis concludes with a summary of all performed research, the conclusions that can be drawn and some suggestions for future research in chapter 7.

## Chapter 2

# Related research

This thesis focusses on the optimization of XQuery evaluation in Pathfinder. As of today, XML is still growing in popularity and many more XML database solutions exist. Two major classes of XML database solutions can be defined:

- native solutions
- relational, non-native solutions

Pathfinder belongs to the latter of the two. Although very different in their approach, research within these two classes of XML database solutions have many similar problems to deal with. A good example of a native system that faces similar research problems as Pathfinder is Natix.

A specific part of XQuery optimization discussed in this thesis involves the optimization of XPath location steps. Location steps in Pathfinder's relational algebra are expressed in terms of the staircase join operator. Chapter 6 discusses the subject of XPath symmetries. A leading research paper on this subject is XPath: Looking Forward.

### 2.1 Natix

Pathfinder evaluates XPath expressions using an algebraic approach. This approach is also adopted by Natix [12], a native XML database system. Compiling expressions into an algebra is expected to yield great optimization possibilities. In Natix, the algebra is based on ordered sequences of tuples, which may be nested. This is a major difference when compared to Pathfinder, that solely uses flat relations of tuples. Natix uses a pipelined approach to evaluate its query plans, where Pathfinder evaluates queries in a sequential manner. Although there are big differences between the Natix and Pathfinder systems, when taking a closer look at their query plans, they turn out to be quite similar in their strategy.

## 2.2 Reverse axis removal

Chapter 6 takes a closer look at the possibilities of recognizing XPath symmetries on the level of the relational algebra. This chapter is inspired by the article XPath: Looking Forward [11] by D. Oltenau et al. In this article, the rare (reverse axis removal) algorithm is presented. The algorithm recognizes XPath expressions containing reverse axis steps (such as **ancestor**, **preceding**) and rewrites them into expressions only containing forward axis steps (such as **descendant**, **following**). XPath expressions containing only forward axis steps can be evaluated by a single scan over the XML document, which is highly desirable in stream based XML applications.

## 2.3 Other systems

As mentioned, today there are many XML database solutions. Three leading projects are Galax, Timber and X-Hive, all native XML solutions. Unlike Pathfinder, they do not store their XML documents in a relational database backend, but are directly built on top of the XML tree structure. Although different in approach, these systems also share some interesting similarities with Pathfinder. A comparison of the four systems has been made in [10]. Pathfinder appears to outperform these systems by quite a big margin.

## Chapter 3

# XQuery to algebra compilation

Evaluating an XQuery input, Pathfinder first compiles the query to its own relational algebra before actually performing the query in the MIL language. This chapter is a prelude to chapter 4 which gives a full specification of Pathfinder’s relational model by giving an introduction to the algebra. An explanation of all operators will be given, followed by a section on the principals of loop lifting, what it comprehends and how it is used in the compilation process. Next, using the Haskell prototype of Pathfinder, a step-by-step example is given on how an XQuery expression is transformed into an algebraic query plan. The chapter concludes with a small section on how to read such a generated query plan.

### 3.1 Introducing the algebraic expressions

As mentioned in the introduction, the algebra used is a quite primitive one, designed to match the capabilities of modern SQL-based systems. It contains all usual operators such as selection, projection and regular joins. An important feature to the algebra is the added staircase join, also previously explained. The full set of operators is listed in figure 3.1.

Where the operators for selection, disjoint union, the cartesian product and the equi-join are self-explanatory, the other operators might need some more explanation. The projection operator  $\pi$  performs a projection with an optional renaming of columns  $b_1, \dots, b_n$  into  $a_1, \dots, a_n$ . To maintain order of elements, which is an important aspect of XQuery,  $\rho$  is used. Using the sort order defined by columns  $a_1, \dots, a_n$ , row numbers are stored in the new column  $b$ . Row numbers start from 1 in each partition defined by the optional grouping parameter  $p$ .  $C \stackrel{\rho}{\alpha::n} D$  performs a staircase join operation along the XPath axis defined by  $\alpha$  with nodetest  $n$  originating in the context nodes  $C$  and producing a result in from the document nodes  $D$ .

$\pi_{a_1:b_1, \dots, a_n:b_n}$	projection (and renaming)
$\sigma_a$	selection
$\delta$	unique
$\dot{\cup}$	disjoint union
$\times$	cartesian product
$\bowtie_{a=b}$	equi-join
$\varrho_{b:(a_1, \dots, a_n)/p}$	row numbering
$\sqcup_{\alpha::n}$	staircase join (axis $\alpha$ , node test $n$ )
$\varepsilon$	element construction
$\odot_{b:(a_1, \dots, a_n)}$	$n$ -ary arithmetic/comparison operator $\cdot$
$\begin{array}{c c} a & b \\ \hline \end{array}$	literal table

Figure 3.1: Operators of the relational algebra,  $a$  and  $b$  being column names.

Another important feature of XQuery is the construction of new elements in the query result. In the algebra this operation is performed by use of the operator  $\varepsilon$ . As this operation only affects the representation of the query result without contributing to the actual query, it will be omitted in the remainder of this thesis. Operator  $\odot_{b:(a_1, \dots, a_n)}$  applies the  $n$ -ary arithmetic or comparison operator  $\cdot$  to columns  $a_1, \dots, a_n$  and stores the result in a new column  $b$ . XQuery expressions evaluate to ordered, finite sequences of items that cannot be nested. In terms of relational tables, a sequence can be represented as a table with two columns: *pos*, an integer sequence providing the required order and a column *item* containing the actual sequence items. An empty sequence is represented by an empty relation. To give an example, the sequence ("a", "b", "c") has relational representation:

<i>pos</i>	<i>item</i>
1	"a"
2	"b"
3	"c"

Although both the database (through the encoding principles of XPath accelerator) and the algebra (added staircase join) are made tree aware, there still remains a major difference between XQuery and relational equivalent used by the RDBMS. Where XQuery is based on an iterative scheme with **for-return** constructs (also known as FLWORS: *For*, *Let*, *Where*, *Order*, *Return*), the relational system (and thus the algebra) relies on non-iterative table joins. Figure 3.2 shows a typical XQuery **for-return** construct. In this figure,  $e[x\$v]$  denotes replacement of all free occurrences of  $\$v$  in  $e$  by  $x$ . To make the compilation of XQuery to the algebra work, [5] introduces *loop lifting*.

```

for $v in (x1, x2, ..., xn) return e ≡
(e[x1\$v], e[x2\$v], ..., e[xn\$v])

```

Figure 3.2: A typical XQuery `for-return` construct.

### 3.2 Loop lifting

The principle of loop lifting is best explained in its simplest form. Consider an XQuery containing the following loop with constant subexpression  $c$ :

```
for $v in (1, 2, 3) return c
```

To compile loops, the following strategy is used [5]:

1. A loop of  $n$  iterations is represented by a relation `loop` with a single column *iter* of  $n$  values  $1, 2, \dots, n$ .
2. If a constant subexpression  $c$  occurs inside a loop body  $e$ , the relational representation of  $c$  is *lifted* (intuitively, this accounts for  $n$  independent evaluations of  $e$ ).

According to the loop lifting strategy, constant subexpression  $c$  will now be lifted by computation of the Cartesian product

$$\text{loop} \times \frac{\text{pos} \mid \text{item}}{1 \mid c}$$

For further evaluation of the query result, the system now solely operates with the loop lifted relation. With an initial `loop` relation of three iterations, the loop lifted relation for the example above is represented by:

$$\begin{array}{c|c} \text{iter} & \\ \hline 1 \\ 2 \\ 3 \end{array} \times \frac{\text{pos} \mid \text{item}}{1 \mid c} = \begin{array}{c|c|c} \text{iter} & \text{pos} & \text{item} \\ \hline 1 & 1 & c \\ 2 & 1 & c \\ 3 & 1 & c \end{array}$$

It gets more difficult when the idea of loop lifting is generalized with respect to arbitrarily nested `for`-loops. Nesting of `for`-loops causes occurrence of more and different *variable scopes*. Figure 3.3 gives an example of nested `for`-loops along with their variable scopes. Here,  $s_{x.y}$ , with  $x \in \{0, 1, \dots\}^*$  and  $y \in \{0, 1, \dots\}$ , denotes the  $y^{\text{th}}$  child scope of scope  $s_x$ . Before explaining loop lifting for arbitrarily nested loops however, a distinction has to be made between two different types of variables: *bound* and *free* variables. In the following,  $q_x(e)$  denotes the relational representation of expression  $e$  in scope  $s_x$ .

$$s \left\{ \begin{array}{l} ( \text{ for } \$v_0 \text{ in } e_0 \text{ return} \\ \quad s_0 \{ e'_0 , \\ \quad \text{ for } \$v_1 \text{ in } e_1 \text{ return} \\ \quad \quad s_1 \left\{ \begin{array}{l} \text{ for } \$v_{1.0} \text{ in } e_{1.0} \text{ return} \\ \quad s_{1.0} \{ e'_{1.0} \end{array} \right. \end{array} \right.$$

Figure 3.3: Nesting and different variable scopes.

### 3.2.1 Loop lifting with bound variables

Loop lifting with bound variables is the less difficult of the two cases and closely resembles the situation with a constant subexpression. Consider the following `for`-loop in its directly enclosing scope:

$$s_x \left\{ \begin{array}{l} \vdots \\ \text{ for } \$v_{x.y} \text{ in } e_{x.y} \text{ return} \\ \quad s_{x.y} \{ e'_{x.y} \\ \quad \vdots \end{array} \right. \quad (3.1)$$

In this example, variable  $\$v_{x.y}$  is bound to each item of the sequence resulting from evaluation of  $e_{x.y}$  in  $s_x$ . Each binding of  $\$v_{x.y}$  will then be used in the evaluation of  $e'_{x.y}$ . To get a suitable representation for  $\$v_{x.y}$  the values of  $q_x(e_{x.y})$  have to be retained. Because  $\$v_{x.y}$  is only visible in scope  $s_{x.y}$ , a new *iter* attribute is added containing consecutive numbers along with a constant attribute *pos* with a value of 1. In terms of the algebra, the representation of  $\$v_{x.y}$  is retained by:

$$q_{x.y}(\$v_{x.y}) = \frac{pos}{1} \times \pi_{iter:inner,item}(\varrho_{inner:(iter,pos)}(q_x(e_{x.y})))$$

Notice the introduction of the *inner* attribute to denote evaluation of the loop in scope  $s_{x.y}$ . The *iter* attribute in the representation of  $e_{x.y}$  may be referred to as *outer*, since it denotes the iterations in the outer loop of scope  $s_x$ . To further evaluate the query comprising the loop above, the relation `loop` needs to be redefined based on  $q_{x.y}(\$v_{x.y})$ :

$$\text{loop}_{x.y} = \pi_{iter}(q_{x.y}(\$v_{x.y})) .$$

**Example 3.1.** Consider the following `for`-loop in its directly enclosing scope  $s$ :

$$s \left\{ \begin{array}{l} \vdots \\ \text{ for } \$v_0 \text{ in } (1, 2, 3) \text{ return} \\ \quad s_0 \{ \$v_0 \\ \quad \vdots \end{array} \right.$$

The sequence  $(1, 2, 3)$  will be evaluated in scope  $s$ . Variable  $\$v_0$  is then bound to each item in the resulting sequence. Evaluation of  $(1, 2, 3)$  in scope  $s$  yields the following representation  $q(1, 2, 3)$ , with `loop` the `loop` relation on the single attribute *iter* describing the iterations of the *outer* loop body  $s$ :

$$\text{loop} \times \begin{array}{c|c} \textit{pos} & \textit{item} \\ \hline 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{array}$$

Bindings of  $\$v_0$  to  $(1, 2, 3)$  are used for evaluation of the expression in scope  $s_0$ . A suitable representation  $q_0$  of  $\$v_0$  in  $s_0$  can be computed by retaining the representation of  $(1, 2, 3)$  and assigning a new *iter* attribute with consecutive numbers and a constant *pos* value of 1:

$$q_0(\$v_0) = \frac{\textit{pos}}{1} \times \pi_{\textit{iter}:\textit{inner},\textit{item}}(\varrho_{\textit{inner}:\langle\textit{iter},\textit{pos}\rangle}(q(1, 2, 3)))$$

This represents the following encoding of variable  $\$v_0$ :

$$\begin{array}{c|c|c} \textit{iter} & \textit{pos} & \textit{item} \\ \hline 1 & 1 & 1 \\ 2 & 1 & 2 \\ 3 & 1 & 3 \end{array}$$

Exiting the loop, the query result is computed by renumbering the items on the order specified by  $\langle\textit{iter}, \textit{pos}\rangle$  and extending it with a new *iter* value of 1:

$$\textit{result} = \frac{\textit{iter}}{1} \times \pi_{\textit{pos}:\textit{pos1},\textit{item}}(\varrho_{\textit{pos1}:\langle\textit{iter},\textit{pos}\rangle}(q_0(\$v_0)))$$

A relational table representation of the result looks like:

$$\begin{array}{c|c|c} \textit{iter} & \textit{pos} & \textit{item} \\ \hline 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 3 & 3 \end{array}$$

□

### 3.2.2 Loop lifting with free variables

Expressions in XQuery may refer to all variables bound in enclosing scopes. In other words, a variable  $\$v_{x.y}$  bound in scope  $s_{x.y}$  may be referred to within all its child scopes. If such a child scope is then viewed in isolation,

the variable  $\$v_{x.y}$  appears to be a *free* variable. Extending 3.1, the situation now looks like:

$$s \left\{ \begin{array}{l} \vdots \\ \text{for } \$v_x \text{ in } e_x \text{ return} \\ \quad s_x \left\{ \begin{array}{l} ( \$v_x, \\ \quad \text{for } \$v_{x.y} \text{ in } e_{x.y} \text{ return} \\ \quad \quad s_{x.y} \{ (\$v_x, \$v_{x.y}) \} \\ \quad ) \end{array} \right. \\ \vdots \end{array} \right.$$

The free variable here is  $\$v_x$ , when taking an isolated view of scope  $s_{x.y}$ . Suppose that  $e_x$  and  $e_{x.y}$  are both sequences of length 2. With the first binding of  $\$v_x$  in the *outer* loop body, two evaluations of the *inner* loop occur. During the next iteration of the outer loop, there will be two more evaluations of the inner loop. Now, a new relation can be defined, representing the semantics of this nested iteration:

<i>outer</i>	<i>inner</i>
1	1
1	2
2	3
2	4

The tuple (2,3) for example indicates that for the third iteration of the inner loop, the outer loop is in its second iteration. This relation maps representations between scopes  $s_x$  and  $s_{x.y}$  and will be called  $\mathbf{map}_{(x.y)}$ . The representation of the free variable  $\$v_x$  in scope  $s_{x.y}$  can now be derived via the equi-join:

$$q_{x.y}(\$v_x) = \pi_{iter:inner,pos,item}(q_x(\$v_x) \bowtie_{iter=outer} \mathbf{map}_{(x.y)})$$

The relation  $\mathbf{map}_{(x,y)}$  can be derived from the representation of the domain  $e_{x.y}$  of variable  $\$v_{x.y}$ :

$$\mathbf{map}_{(x,y)} = \pi_{outer:iter,inner}(q_{inner:(iter,pos)}(q_x(e_{x.y}))) .$$

**Example 3.2.** Consider the following example, which is an extended version of example 3.1:

$$s \left\{ \begin{array}{l} \vdots \\ \text{for } \$v_0 \text{ in } (1,2) \text{ return} \\ \quad s_0 \left\{ \begin{array}{l} ( \$v_0, \\ \quad \text{for } \$v_{0.0} \text{ in } (10,20) \text{ return} \\ \quad \quad s_{0.0} \{ (\$v_0, \$v_{0.0}) \} \\ \quad ) \end{array} \right. \\ \vdots \end{array} \right.$$

A second loop has been added within scope  $s_0$ . Within this loop, variable  $\$v_0$  plays the role of *free* variable. Referring to the first loop as *outer* and the second loop as *inner*, *outer* can be mapped onto *inner* by using the relation:

<i>outer</i>	<i>inner</i>
1	1
1	2
2	3
2	4

For each *outer*-loop iteration, two iterations of the *inner*-loop occur, resulting in a total of  $2 \times 2 = 4$  iterations of *inner*. The *map*-relation is referred to as  $\text{map}_{(0,0,0)}$ , as it maps between the scopes  $s_0$  and  $s_{0,0}$ . Within scope  $s_{0,0}$ , variable  $\$v_{0,0}$  is dealt with the same way as  $\$v_0$  in example 3.1. To retain the representation of  $\$v_0$  in  $s_{0,0}$   $\text{map}_{0,0,0}$  is used. For example, when *inner* is in its second iteration,  $v_{0,0}$  will be bound to 20, while according to  $\text{map}_{0,0,0}$  *outer* is in its first iteration and  $\$v_0$  will be bound to 1. Now, the intermediate result in  $s_{0,0}$  can be made up:

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	1
1	2	10
2	1	1
2	2	20
3	1	2
3	2	10
4	1	2
4	2	20

This result is mapped back onto scope  $s_0$ , the *iter* attribute represents the  $i^{\text{th}}$  iteration of the *inner*-loop and is mapped back to the *outer*-loop using  $\text{map}_{0,0,0}$ , and *pos* is renumbered with respect to the new value of *iter*:

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	1
1	2	10
1	3	1
1	4	20
2	1	2
2	2	10
2	3	2
2	4	20

Within the *outer*-loop, apart from  $(\$v_0, \$v_{0,0})$  of the *inner*-loop, another term is added to the result:  $\$v_0$ . This is a bound variable and already

knowing what to do in such a case (see example 3.1) it is clear to see where it fits in the intermediate result. Inserting the representation of  $v_0$  yields:

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	1
1	2	1
1	3	10
1	4	1
1	5	20
2	1	2
2	2	2
2	3	10
2	4	2
2	5	20

As in example 3.1, last thing to do is to exit the *outer*-loop by renumbering all *items* on the order specified by  $\langle iter, pos \rangle$  and adding a new *iter* value of 1. The query result now will be:

<i>iter</i>	<i>pos</i>	<i>item</i>
1	1	1
1	2	1
1	3	10
1	4	1
1	5	20
1	6	2
1	7	2
1	8	10
1	9	2
1	10	20

□

### 3.3 Loop-lifted staircase join

The staircase join operator has been designed for efficient handling of XPath axes. All the XPath location steps are rooted in a *single* sequence of context nodes. However, in XQuery, XPath expressions may occur embedded inside arbitrarily nested for-loops. Such embedded expressions are rooted in *multiple* sequences of context nodes, one for each iteration. This causes the staircase join as described in the introduction to undertake repeated scans over the document table, which has a severe impact on query performance.

To overcome this problem, the loop-lifted staircase join is proposed. It exploits the tree properties of the document table to reduce the number of repeated executions of XPath expressions and evaluates those in a single

scan over the document table. Performance is strongly improved opposed to repeated execution of the "simple" staircase join. Empirical evidence [10] shows a performance improvement up to a factor 10.

The staircase join takes a context relation  $C$  and a document relation  $D$  as its inputs and outputs a relation of the same form as the context relation  $C$  (and may thus serve as input for another staircase join operation). The context relation is a table with two columns *iter* and *item*. The *iter* column describes the different iterations (see the section on loop lifting above), while *item* contains the *preorder* ranks of the context nodes. In the case of a "simple" staircase join, the *iter* column has a constant value of 1. The document relation is a relational table containing all document elements, which are described by the columns *pre*, *size*, *level*, *kind*, *prop* and *frag*. Columns *pre*, *size* and *level* represent a node's *preorder* rank, the size of the subtree it contains and the document level in which it resides. Together, they determine the *postorder* rank of the node. Column *kind* describes the nodetype (e.g. *element*), *prop* contains the node's value (e.g. *element name*) and *frag* contains the fragment number (e.g. the document fragment containing the node). Notice that the context relation only contains a *preorder* rank to describe the context node. To evaluate an axis step, the staircase join first extracts information on the context nodes by looking up their *preorder* rank within the document relation.

As said above, the output relation of a staircase join operation has the same form as the input context relation. A tuple  $(iter, item)$  of the context relation results in zero or more  $(iter, item)$  tuples in the output relation with equal *iter* values and output *items* containing the *preorder* ranks of the result nodes of the axis step originating in the context nodes.

### 3.4 Query plan construction

This section presents two examples on query plan construction. The first example explains the compiler steps using a very basic XPath expression. The query of this example is then extended to support loop lifting in the next example.

**Example 3.3.** This example shows the steps that are taken by the XQuery compiler to translate an XQuery expression into the relational algebra specified in the previous sections.

**Step 1.** The first step is to define the XQuery expression to feed the compiler:

```
doc("auction.xml")/descendant::person
```

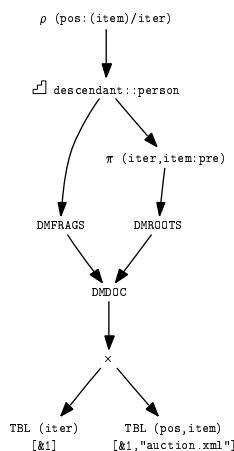


Figure 3.4: Algebraic query plan of example 3.3

The expression uses the document *auction.xml*<sup>1</sup> and outputs all *person* nodes that are descendants of the root.

**Step 2.** Knowing the expression, it now has to be translated into XQuery Core. XQuery Core is a simplified subset of the XQuery grammar, only using simple *for* expressions. The XQuery Core dialect supported by Pathfinder's Haskell prototype can be found in Appendix B. The Core expression of the query in step 1 reads:

```

XPATH (XFNDOC (XSTR "auction.xml")
        (Descendant, XMLElem, ["person"]))

```

**Step 3.** The Core expression of step 2 can now be compiled into a relational expression. This compilation process is built upon a set of compilation rules. Interested readers are encouraged to consult [5] for a detailed description on these rules. The algebraic expression (or query plan) can be represented in a Directed A-cyclic Graph (DAG), see figure 3.4 for the DAG representation of the query used in this example.

**Step 4.** The algebraic expression is used to compute the result<sup>2</sup>. □

Figure 3.4 shows the DAG representation of the algebraic expression of example 3.3. The nodes represent the algebraic operations and relations used. The plan is to be read bottom up, which means that the top node

<sup>1</sup>This is the same document as used by XMark [9], an XML benchmark suite, used to benchmark XQuery performance. Appendix A shows the DTD *auction.xml* complies to.

<sup>2</sup>Between steps 3 and 4, the compiler performs an additional operation called Common Sub-expression Elimination (CSE) to remove duplicate sub-trees from the DAG.

yields the actual query result. The meaning of the nodes defining the algebraic operations should be clear, whereas the remaining nodes need some further explanation:

- TBL (a,b) defines a relation with columns (a,b) and its tuples
- DMDOC represents the document relation (all documents that are addressed by the query)
- DMFRAGS represents all document fragments within the document relation
- DMROOTS represents the roots of the documents in the document relation

**Example 3.4.** As explained in the sections on loop lifting above, loop lifting occurs when the input XQuery expression contains a for-loop. To illustrate the process of loop lifting in an algebraic query plan, the previous example is extended to:

```
for $b in doc("auction.xml")/descendant::person
return $b/child::name
```

This query translates to XQuery core as:

```
XFOR "b" (XPATH (XFNDOC (XSTR "auction.xml"))
           (Descendant, XMLElement, ["person"]))
        (XPATH (XVAR "b") (Child, XMLElement, ["name"]))
```

After compilation and performing CSE, the compiler puts out the query plan of figure 3.5. The bottom part of the DAG is the same as in example 3.3. Only here, after performing a staircase join to fetch all *person* descendants of the root, a loop is entered to retrieve the *name* child of each person. The result of the first axis step (`descendant::person`) is extended with a *pos* and an *inner* attribute by performing two numbering operations. The *inner* attribute distinguishes the inner and outer loop as explained before. For each iteration (*inner* value) and context node pair, axis step `child::name` is performed by means of a loop-lifted staircase join. The resulting relation is then joined with the intermediate result obtained before the loop, based on the values of *outer* and *inner*. □



## Chapter 4

# Pathfinder's relational algebra

Optimization of the relational algebra is largely based on equivalence rules. In order to compose these rules, the algebraic operators have to be specified first. Since these operations only have meaning on a certain relational structure, this structure is also to be defined. In other words, Pathfinder's relational data model has to be charted. The following sections describe both the structure and algebra in detail.

### 4.1 Relational structure

To specify Pathfinder's relational structure, a relation will be represented as a multi-set<sup>1</sup> of tuples. Each relation is defined on a certain domain.

**Definition 4.1.** A multi-set is a set of elements allowing duplicates, not specifying any order on its elements.  $\square$

**Definition 4.2.** A domain  $A$  is a set of atomic values.  $\square$

All values considered atomic within pathfinder are depicted in figure 4.1

**Definition 4.3.** A relation  $R$  is a multi-set of tuples ( $r \mapsto m(r)$ ), with  $r$  a record in the relation and  $m(r)$  a function on  $r$  denoting the multiplicity of  $r$  in  $R$ :  $m : r \rightarrow \mathbb{N}^+$ ,  $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$ . A record  $r$  is defined as a list of attributes  $\langle a_1, \dots, a_n \rangle$ , each defined on its own domain  $dom(a_i)$ . The domain of relation  $R$ ,  $dom(R)$ , is defined as  $dom(R) = dom(a_1) \times \dots \times dom(a_n)$ . For two records  $r_1 \in R_1$  and  $r_2 \in R_2$ , the *concatenation*  $r_1 \oplus r_2$  is defined as the concatenation of the attributes of  $r_1$  and  $r_2$  having domain  $dom(R_1) \oplus dom(R_2)$ .  $\square$

**Definition 4.4.** To accommodate a full specification of all algebraic operators in the next section two additional concepts need to be introduced. The

---

<sup>1</sup>In the remainder of this thesis the terms set and multi-set are used interchangeably. Unless made explicit, a multi-set is intended.

Value	Type
int	Integer
str	String
bool	Boolean
nat	Nonnegative Integer
dec	Float
dbl	Double
node	<i>preorder</i> rank: Integer

Figure 4.1: Atomic values in Pathfinder.

records  $r$  with attributes  $\langle a_1 \dots a_n \rangle$  in a relation  $R$  can be grouped on certain attributes  $a_i \dots a_j$ , with  $i, j \in \{1 \dots n\}$  and  $i \leq j$ . Such a partitioning of  $R$  is denoted as  $[R]_{a_i \dots a_j}$ , formally defined by:

$$[R]_{a_i \dots a_j} = \{ \{ R' \mid R \in \mathcal{P}(R) \wedge \exists \langle v_i, \dots, v_j \rangle (\forall r \in R' : r.a_i = v_i \wedge \dots \wedge r.a_j = v_j) \} \}$$

Here,  $\mathcal{P}(R)$  is the *power* multi-set of  $R$ , i.e. the set of all possible *sub*multi-sets of  $R$ . The record  $\langle a_i \dots a_j \rangle$  consisting only of the partitioning attributes and thus unique for each partition is written as  $\hat{S}_{a_i \dots a_j}$ , with  $S \in [R]_{a_i \dots a_j}$ .

The ordering of a relation is thus far not taken into account as it violates the definition of a multi-set, see definition 4.1. This is a problem as Pathfinder ensures document order in a relation at all times. Therefore, a second concept is introduced:  $\tilde{R}_a$ , a list representation of all records in  $R$  in the order specified by one or more attributes  $a$ . Within  $\tilde{R}_a$ , records might be accessed on their position (row number in a table representation) using  $\tilde{R}_a[i]$ , with  $i = 1 \dots |\tilde{R}_a|$ . Note that  $|\tilde{R}_a| = \text{sum}\{m(r) \mid r \in R\}$ .  $\square$

These two concepts yield two seemingly trivial but important properties:

$$\bigcup_{a_i \dots a_j} [R]_{a_i \dots a_j} = R \quad (4.1)$$

$$\{\tilde{R}_a[1], \dots, \tilde{R}_a[|\tilde{R}_a|]\} = \tilde{R}_a \quad (4.2)$$

Basically, this means that the union of all partitions of a relation  $R$  on attributes  $a_i \dots a_j$  produces the original relation  $R$  and that all individual records of  $\tilde{R}_a$  taken together is equal to  $\tilde{R}_a$  itself.

## 4.2 Specifying the algebra

The structure given in the previous section forms the basis upon which the specification of the algebra operators is built.

**Definition 4.5.** The relational algebra defines operations on relations. These operations define relational expressions. The following constructs are relational expressions:

- *Projection* (and renaming) of  $R$  on attributes  $a_1 \dots a_n$  results in a new relation defined on  $dom(a_1) \times \dots \times dom(a_n)$ :

$$\begin{aligned} \pi_{a_1 \dots a_n}(R) = \{ & | \langle a_1 = r.a_1 \dots a_n = r.a_n \rangle \mapsto k \} | \\ & R' \in [R]_{a_1 \dots a_n} \wedge r = \hat{R}'_{a_1 \dots a_n} \wedge k = |R'| \} \end{aligned}$$

The projection  $\pi_{b:a}$  will project on attribute  $a$  and subsequently rename this attribute to  $b$ .

- *Row numbering* using  $\varrho_{pos:a/b}$  will append the input records with a new attribute  $pos$  containing row numbers as defined by the ordering  $a$ , optionally partitioned on attribute  $b$ .

$$\begin{aligned} \varrho_{pos:a/b}(R) = \{ & | (r \oplus \langle pos = p \rangle \mapsto m(r)) | \\ & R' \in [R]_b \wedge p \in 1 \dots |R'| \wedge r = \tilde{R}'_a[p] \} \end{aligned}$$

Note that there is only numbering on one attribute  $a$  here. Numbering on more than one attribute requires a more complex definition, which is beyond the scope of this document and therefore omitted.

- The join  $R_1 \bowtie_{a_i, b_j} R_2$  produces a selection on the cartesian product of  $R_1$  and  $R_2$ , having domain  $dom(R_1) \oplus dom(R_2)$ :

$$\begin{aligned} R_1 \bowtie_{a_i, b_j} R_2 = \{ & | (r \oplus q \mapsto m_{R_1}(r) \cdot m_{R_2}(q)) | \\ & r \in R_1 \wedge q \in R_2 \wedge r.a_i = q.b_j \} \end{aligned}$$

- The *unique* operator sets all multiplicities of the records in a relation to one:

$$\delta(R) = \{ | (r \mapsto 1) | r \in R \}$$

- The *selection* operator  $\sigma_{a_i}$  selects all records of a relation where attribute  $a_i$  is set to *True*, with  $dom(a_i) = \{True, False\}$ :

$$\sigma_{a_i}(R) = \{ | (r \mapsto m(r)) | r \in R \wedge r.a_i = True \}$$

- Operation of  $\odot_{b:(a_1, \dots, a_n)}$  on relation  $R$  applies the  $n$ -ary arithmetic or comparison operator  $\cdot$  to attributes  $a_1, \dots, a_n$  and extends the input records with result attribute  $b$ , producing a relation  $R'$  with  $dom(R') = dom(R) \oplus dom(b)$ :

$$\odot_{b:(a_1, \dots, a_n)}(R) = \{ | (r \oplus \langle b \rangle \mapsto m(r)) | r \in R \wedge b = r.a_1 \dots \cdot r.a_n \}$$

- The *staircase join* takes a context relation  $C$  and a document relation  $D$ , producing a result from  $D$  originating in  $C$  along axis step  $\alpha$  with nodetest  $n$ .

$$C \underset{\alpha::n}{\bowtie} D = \{ | (\langle iter = c'.iter, item = d.pre \rangle \mapsto m_C(c') \times m_D(d)) \mid \\ c' \in C \wedge d \in D \wedge \exists c \in D : c.pre = c'.item \wedge \\ cond_{\alpha::n} \} \}$$

The term  $cond_{\alpha::n}$  denotes a condition based on nodetest  $n$  (an equality test on the properties *kind* and *prop*, where  $kind = node \mid document \mid node \mid element \mid text$  and *prop* is the element name) and axis step  $\alpha$  that specifies which variant of the staircase join will be used, imposing one of the conditions depicted in figure 4.2.<sup>2</sup> Note that for clarity *post* is used to specify a node's *postorder* rank instead of writing out  $size + pre - level$  (which is how *post* is computed).

The multiplicity of each record in the result is equal to the product of the multiplicities of  $c'$  and  $d$  in sets  $C$  and  $D$  respectively. Since all elements  $d$  in the document relation are uniquely defined by their *pre* attribute, the product will be equal to  $m_C(c')$ . It is safe to assume that in general every  $(iter, item)$  record in  $C$  will be unique. Therefore, in the remainder of this thesis, the multiplicity of every result record of the staircase join is assumed to be equal to one. This is solely done for the purpose of readability and should be replaced with the product of multiplicities to be universally applicable.

Note that there are no separate definitions for the staircase join and its loop-lifted version. There is only a difference in computation algorithm, which is chosen based on the values of  $c'.iter$ . If  $c'.iter$  has a constant value of 1, the "simple" staircase join is chosen, otherwise it is better to take the loop-lifted approach.

□

### 4.3 Rewriting expressions

As said, rewriting algebraic expressions is largely based on the use of equivalence rules. The following expression frequently occurs in algebraic plans, as a result of Pathfinder's compilation process:

$$\pi_{iter, item}(\rho_{pos:\langle item \rangle / iter}(R))$$

---

<sup>2</sup>The *following-sibling* and *preceding-sibling* axes are omitted here as they impose an additional condition relating to the context node's parent node and require a more complex definition that is beyond the scope of this thesis.

Axis step $\alpha$	Imposed conditions
self	$d.pre = c.pre$
parent	$d.pre < c.pre \wedge d.post < c.post \wedge d.level = d.level - 1$
child	$d.pre > c.pre \wedge d.post > c.post \wedge d.level = d.level + 1$
ancestor	$d.pre < c.pre \wedge d.post > c.post$
ancestor-or-self	$(d.pre < c.pre \wedge d.post > c.post) \vee d.pre = c.pre$
descendant	$d.pre > c.pre \wedge d.post < c.post$
descendant-or-self	$(d.pre > c.pre \wedge d.post < c.post) \vee d.pre = c.pre$
following	$d.pre > c.pre \wedge d.post > c.post$
preceding	$d.pre < c.pre \wedge d.post < c.post$

Figure 4.2: Imposed conditional properties by axis step  $\alpha$ .

The numbering operator is used to append all records in  $R$  with a  $pos$  attribute, which is subsequently projected out by the projection  $\pi_{iter,item}$ . This leads to a first equivalence rule:

$$\pi_{iter,item}(\rho_{pos:\langle item \rangle/iter}(R)) = \pi_{iter,item}(R) \quad (4.3)$$

**Proof** The equivalence seems trivial, but there is more to it than it seems when writing it out:

$$\pi_{iter,item}(\rho_{pos:\langle item \rangle/iter}(R))$$

Using the specifications of  $\pi$  and  $\rho$  of the previous section results in:

$$\begin{aligned} &= \{ | (\langle iter = s.iter, item = s.item \rangle \mapsto k) | \\ &\quad S \in [\{ | (r \oplus \langle pos : p \rangle \mapsto m(r)) | R' \in [R]_{iter} \wedge \\ &\quad\quad p \in 1 \dots |R'| \wedge r = \tilde{R}'_{item}[p] | \}]_{iter,item} \\ &\quad \wedge s = \hat{S}_{iter,item} \wedge k = |S| | \} \end{aligned}$$

Since the projection  $\pi_{iter,item}$  makes the added  $pos$  attribute by  $\rho_{pos:\langle item \rangle/iter}$  obsolete, it can be removed from the equation without affecting the result:

$$\begin{aligned} &= \{ | (\langle iter = s.iter, item = s.item \rangle \mapsto k) | \\ &\quad S \in [\{ | (r \mapsto m(r)) | R' \in [R]_{iter} \wedge \\ &\quad\quad p \in 1 \dots |R'| \wedge r = \tilde{R}'_{item}[p] | \}]_{iter,item} \\ &\quad \wedge s = \hat{S}_{iter,item} \wedge k = |S| | \} \end{aligned}$$

Making use of properties 4.1 and 4.2, it becomes clear that relation  $S$  is nothing more than an element of  $[R]_{iter,item}$ :

$$\begin{aligned} &= \{ | (\langle iter = s.iter, item = s.item \rangle \mapsto k) | \\ &\quad S \in [R]_{iter,item} \wedge s = \hat{S}_{iter,item} \wedge k = |S| | \} \end{aligned}$$

Renaming of  $S$  to  $R$  yields:

$$\begin{aligned}
 &= \{ | (\langle iter = r.iter, item = r.item \rangle \mapsto k) \mid \\
 &\quad R' \in [R]_{iter, item} \wedge r = \hat{R}'_{iter, item} \wedge k = |R'| \} \\
 &= \pi_{iter, item}(R)
 \end{aligned}$$

This proves that the equivalence of 4.3 is indeed valid. □

# Chapter 5

## Staircase join

The previous chapter ended with an example on how to compose equivalence rules for algebraic expressions. In this thesis, the focus lies on the staircase join operator, that evaluates XPath expressions. This chapter concentrates on equivalence rules that involve the staircase join. Apart from the staircase join described before, two other variants have already been proposed in [8]. The staircase join implemented in Pathfinder is called the *right* staircase join, the two other variants are called *left* and *generalized* staircase join. All three staircase joins are taken into account in this research.

### 5.1 Staircase join variants

In chapter 4, the staircase join has been defined by:

$$C \stackrel{\alpha::n}{\bowtie} D = \{ | ((iter = c'.iter, item = d.pre) \mapsto 1) | \\ c' \in C \wedge d \in D \wedge \exists c \in D : c.pre = c'.item \wedge cond_{\alpha::n} | \}$$

The staircase join takes two relations as its inputs, relation  $C$  contains all context nodes, relation  $D$  is the document relation. The context relation  $C$  is defined on attributes  $iter$ ,  $item$ , document relation  $D$  contains all node information and is defined on attributes  $pre$ ,  $size$ ,  $level$ ,  $kind$ ,  $prop$  and  $frag$ . For each node in the context set  $C$ , the staircase join evaluates axis step  $\alpha$ , with a possible node test defined by  $n$ . The result is produced from document relation  $D$ . Conditions imposed by  $\alpha :: n$  have already been depicted in figure 4.2.

**Definition 5.1.** Three staircase join variants are proposed in [8], corresponding to the three major classes of XPath expressions:

- The *right* staircase join is specified for XPath expressions without predicates. These expressions are of the form  $s_1/s_2/\dots/s_n$ , where  $s_i = \alpha :: n$  denotes an axis step along axis  $\alpha$  and node test  $n$ . The

output sequence of each step  $s_i$  serves as input for  $s_{i+1}$ . An ordinary join would output a concatenation of matching records from  $C$  and  $D$ , but since records  $c \in C$  are irrelevant for  $s_{i+1}$ , they are discarded:

$$C \xrightarrow{\alpha::n} D = \{ | (\langle iter = c'.iter, item = d.pre \rangle \mapsto 1) | \\ c' \in C \wedge d \in D \wedge \exists c \in D : c.pre = c'.item \wedge cond_{\alpha::n} \}$$

- XPath expressions might include path expressions within predicates: a step  $s_i$  of the form  $\alpha :: n[p]$ , where  $p$  is a relative path expression of the form  $./s'_1/s'_2/\dots/s'_n$ . The *left* staircase join deals with predicates containing single step path expressions:  $/s_1[./s_2]$  is an example of such an expression. Because  $s_2$  resides in a predicate, the output for a join of  $s_2$  between the result of  $s_1$  and the document relation should produce nodes from the context of  $s_2$ :

$$C \xleftarrow{\alpha::n} D = \{ | (c' \mapsto 1) | \\ c' \in C \wedge d \in D \wedge \exists c \in D : c.pre = c'.item \wedge cond_{\alpha::n} \}$$

- More complex expressions might have predicates containing more than one path expression, such as:  $s_1[./s_2/s_3]$ . Both *left* and *right* staircase joins not suffice in this situation. It is desirable to have a join that outputs information of both joined records. This is done by the *generalized* staircase join:

$$C \xleftrightarrow{\alpha::n} D = \{ | (\langle iter = c'.iter, c.item = c'.item, d.item = d.pre \rangle \mapsto 1) | \\ c' \in C \wedge d \in D \wedge \exists c \in D : c.pre = c'.item \wedge \\ cond_{\alpha::n} \}$$

The *item* attributes in the output relation are prefixed with either a lower case 'c' or 'd', corresponding to the input relations  $C$  and  $D$  respectively.

□

Both left and right staircase joins can be expressed in terms of the generalized staircase join:

$$C \xrightarrow{\alpha::n} D = \pi_{iter,item:d.item}(C \xleftrightarrow{\alpha::n} D) \quad (5.1)$$

$$C \xleftarrow{\alpha::n} D = \pi_{iter,item:c.item}(C \xleftrightarrow{\alpha::n} D) \quad (5.2)$$

## 5.2 Exploring possible equivalence rules

This section explores possible equivalence rules for the staircase join. The followed strategy is first to investigate if there are any commutative and/or associative properties. Based on the results obtained, other possibilities are explored. Finally, it is investigated if certain operations (e.g. selection, projection) might be pushed through a staircase join.

### 5.2.1 Commutativity

The definition of commutativity states that:

**Definition 5.2.** Two elements  $x$  and  $y$  of a set  $S$  are said to be commutative under a binary operation  $*$  if they satisfy  $x * y = y * x$ .  $\square$

Taking  $*$  to be the generalized staircase join and  $x$  and  $y$  to be respectively a context relation  $C$  and document relation  $D$ , commutativity for the staircase join implies that:

$$C \underset{\alpha::n}{\sqcup} D = D \underset{\alpha::n}{\sqcup} C \quad (5.3)$$

However, by definition, the context operand of the staircase join is a relation only containing (apart from the *iter* value) a sequence of *preorder* ranks (*item*). To perform a staircase join, all other node properties (*size*, *level*, etc.) are extracted from the document operand based on (*item,pre*): for every  $c \in C$  there must be a  $d \in D$  having  $c.item = d.pre$ . Should the staircase join support commutativity, then also, for every  $d \in D$  there should be a  $c \in C$  having  $d.item = c.pre$ . As document relation  $D$  has no *item* attribute defined, this is in violation of the definition. Therefore, the generalized staircase join does not support commutativity. It is obvious that this is true for all three staircase join variants.

### 5.2.2 Associativity

The definition of associativity states that:

**Definition 5.3.** Three elements  $x$ ,  $y$  and  $z$  of a set  $S$  are said to be associative under a binary operation  $*$  if they satisfy  $(x * y) * z = x * (y * z)$ .  $\square$

Taking  $*$  to be the generalized staircase join,  $x$  a context relation  $C$  and  $y$  and  $z$  to be two instances<sup>1</sup> of the document relation  $D$ :  $D_1$  and  $D_2$ , associativity for the staircase join implies that:

$$(C \underset{\alpha::n}{\sqcup} D_1) \underset{\alpha::n}{\sqcup} D_2 = C \underset{\alpha::n}{\sqcup} (D_1 \underset{\alpha::n}{\sqcup} D_2) \quad (5.4)$$

---

<sup>1</sup>Note that this thesis restricts to single document queries only.

However, by definition the right hand side of the equation does not describe a valid expression. The staircase join is defined on a context relation and a document relation. The context relation is defined on records  $\langle iter, item \rangle$  of which *iter* describes one or more iterations.  $D_1$  and  $D_2$  are both document relations without *iter* attributes. Therefore, equation 5.4 is in violation with the definition of the staircase join. It is obvious that neither of the three staircase join variants supports associativity.

### 5.2.3 Asymmetric staircase join

Above, it has been shown that both commutative and associative properties do not hold for the staircase join. This is caused by the fact that the staircase join is defined as an *asymmetric* operator: its operands are defined on different domains. Redefining the staircase join on operands defined on equal domains (*symmetric*) imposes a reinvestigation of its commutative properties. Furthermore, if the output relation is also redefined on the same domain as the two input relations, equation 5.4 will describe a valid expression and associativity seems plausible.

## 5.3 Symmetric staircase join

The currently implemented asymmetric staircase join is defined on an explicit order of its operands. The left operand should describe a context relation  $C$  on records  $\langle iter, item \rangle$ , whereas the right operand should describe the document relation  $D$ , i.e. all document nodes and their properties. Only the left hand side may carry an *iter* attribute. The context relation  $C$  is dependent of the document relation  $D$ , based on values of *item* and *pre*. During evaluation of an axis step, the staircase join performs a lookup in  $D$  for every node in context  $C$  to extract all node properties.

Due to these restrictions, the left and right operand cannot be freely interchanged. The symmetric staircase join should support:

- *iter* values from either operand
- 'independent' operands  $C$  and  $D$

To meet these requirements the operands should have identical domains, defined on attributes *iter*, *pre*, *size level*, *kind*, *prop* and *frag*. Note that when renaming the *item* attribute of the context relation to *pre* this domain corresponds to:  $dom(C) \times dom(D)$ . The output relation should also be defined on this domain as it might serve as input for a subsequent staircase join operation.

Leaving the *iter* values out of consideration, only focussing on the independence of operands  $C$  and  $D$ , a first redefinition of the (generalized)

staircase join is made by:

$$C \stackrel{\sqcup}{\underset{\alpha::n}{\bowtie}} D = \{ | (\langle iter = c.iter \rangle \oplus (\triangleleft_c (\tilde{\pi}_{iter}(c))) \oplus (\triangleleft_d d) \mapsto 1) | \\ c \in C \wedge d \in D \wedge cond_{\alpha::n} \}$$

Relations  $C$  and  $D$  are both defined on attributes *pre*, *size*, *level*, *kind*, *prop* and *frag*, where  $C$  carries an additional attribute *iter*. Operation  $\tilde{\pi}_{iter}$  is used to project out the *iter* attribute. It serves as an abbreviation of projecting on all other attributes of the relation. Prefixing operator  $\triangleleft$  is used to prefix all context and document attributes with  $c$ . and  $d$ . respectively. Prefixed attributes might be unprefixing by using the unprefixing operator  $\triangleright$ . The conditions  $cond_{\alpha::n}$  remain unaffected.

This definition meets one of the two requirements of the symmetric staircase join as made up above. Context relation  $C$  does not anymore depend on the document relation  $D$  for looking up node information of the context nodes it contains. Providing support for *iter* attributes from either operand is more complicated.

Chapter 3 has already explained the raison d'être of the *iter* attribute. XQuery queries might contain (nested) for-loops. In compilation, loops are lifted to maximize performance by minimizing intermediate result sizes. To keep track of all iterations, a loop relation is introduced. The loop relation contains a single attribute *iter* with values  $1..n$  describing a loop of  $n$  iterations.

In its initial design, the staircase join did not embody any logic to efficiently evaluate XPath axis steps embedded in for-loops. These expressions are rooted in multiple sequences of context nodes, one sequence for each iteration, and were therefore evaluated in multiple scans over the document table. As described in section 3.3 a loop-lifted version of the staircase join has been introduced to overcome this problem and maximize performance. To keep track of all iterations, the staircase join also uses the loop relation, which is embodied in the *iter* attribute of the input context. A staircase join without loop lifting is called a simple staircase join operation and can be expressed as a loop-lifted staircase join on a constant *iter* value of  $1^2$ .

Recapitulating, the goal of the symmetric staircase join is to be able to freely interchange its operands. In its current form, the staircase join only supports an *iter* value from its left operand. Additionally, the staircase join should support *iter* values from its right operand or *iter* values from *both* operands. Although loop lifting is not always necessary, Pathfinder always maintains a loop relation. If no loop lifting occurs, the *iter* value is set to a constant value of 1. Therefore, it is not necessary to define a case where neither of the operands of the staircase join carries an *iter* attribute.

---

<sup>2</sup>Actually, in the implementation of the staircase join the simple and loop-lifted versions are two separate algorithms. The simple staircase join algorithm has some optimizations not applicable to the loop-lifted version.

Based on the three possible cases the symmetric staircase join on operands  $C$  and  $D$  should:

- ***iter* attribute only in  $C$** : evaluate the axis step for each combination of  $c \in C$  and  $d \in D$  and append the result with  $c.iter$ .
- ***iter* attribute only in  $D$** : evaluate the axis step for each combination of  $c \in C$  and  $d \in D$  and append the result with  $d.iter$ .
- ***iter* attribute in both  $C$  and  $D$** : only evaluate the axis step for  $c \in C$  and  $d \in D$  if  $c.iter = d.iter$ ; the result should be appended with either  $c.iter$  or  $d.iter$ .

Since the symmetric staircase join does not distinguish between the left and right operand, it defines no explicit context and document relations. Therefore, from this point, the symmetric staircase join operands will be called  $X$  and  $Y$ , to prevent confusion. Extending the first definition with conditions on *iter* to distinguish between the three cases sketched above, the complete definition of the symmetric staircase join reads:

**Definition 5.4.** For two relations  $X$  and  $Y$ , the general symmetric staircase join is defined on  $x \in X$  and  $y \in Y$  as:

$$X \overset{\rightarrow}{\bowtie}_{\alpha::n} Y = \{ | ((iter = \max\{x.iter, y.iter\}) \oplus (\underset{x}{\triangleleft} (\tilde{\pi}_{iter}(x))) \oplus (\underset{y}{\triangleleft} (\tilde{\pi}_{iter}(y))) \mapsto 1) | \\ x \in X \wedge y \in Y \wedge \text{cond}_{\alpha::n} \wedge \text{cond}_{x.iter, y.iter} | \}$$

$$\text{cond}_{x.iter, y.iter} = ((x.iter = y.iter) \vee (x.iter \neq 0 \wedge y.iter = 0) \vee (x.iter = 0 \wedge y.iter \neq 0)) \wedge \neg(x.iter = 0 \wedge y.iter = 0)$$

The function  $\max$  is used to select the right *iter* attribute. Since  $x.iter$  is equal to  $y.iter$  or at most one of them is equal to zero, the maximum of the two will describe the right value. Conditions  $\text{cond}_{\alpha::n}$  remain unaffected, only here, relations  $C$  and  $D$  are replaced with  $X$  and  $Y$  respectively. The additional conditions  $\text{cond}_{x.iter, y.iter}$  are to distinguish between the possible  $(x.iter, y.iter)$  combinations. A truth table is depicted in figure 5.3. The table uses sample values 0 and 1. These values may be replaced with either integer greater than zero.  $\square$

The symmetric staircase join can also be expressed into a right and left variant by:

- Right staircase join:

$$X \overset{\rightarrow}{\bowtie}_{\alpha::n} Y = \{ | ((iter = \max\{x.iter, y.iter\}) \oplus (\tilde{\pi}_{iter}(y)) \mapsto 1) | \\ x \in X \wedge y \in Y \wedge \text{cond}_{\alpha::n} \wedge \text{cond}_{x.iter, y.iter} | \}$$

In terms of the general staircase join:

$$X \xrightarrow{\alpha::n} \sqcup Y = \tilde{\pi}_{x.*}(X \sqcup_{\alpha::n} Y)$$

- Left staircase join:

$$X \xleftarrow{\alpha::n} \sqcup Y = \{ | (\langle iter = \max\{x.iter, y.iter\} \rangle \oplus (\tilde{\pi}_{iter}(x)) \mapsto 1) | \\ x \in X \wedge y \in Y \wedge \text{cond}_{\alpha::n} \wedge \text{cond}_{x.iter, y.iter} | \}$$

In terms of the general staircase join:

$$X \xleftarrow{\alpha::n} \sqcup Y = \pi_{y.*}(X \sqcup_{\alpha::n} Y)$$

## 5.4 Commutativity and associativity revisited

Redefining the staircase join to a symmetric version puts the investigation of its commutative and associative properties into another perspective. Both operands and its output are defined on the same domain. Therefore, equations 5.3 and 5.4 denote valid expressions for the symmetric staircase join. This section revisits the concepts of commutativity and associativity with the new staircase join definition.

### 5.4.1 Commutativity

Already, it has been said that equation 5.4 denotes a syntactically correct expression according to the symmetric staircase join definition. Semantically however, the equation remains invalid. Taking  $\alpha$  to be the **descendant** axis and  $n$  an arbitrary element, the equation becomes:

$$X \sqcup_{\text{desc}::*} Y = Y \sqcup_{\text{desc}::*} X$$

The left hand side describes all  $x \in X$  that have descendants  $y \in Y$ , whereas the right hand side describes all  $y \in Y$  that have descendants  $x \in X$ . Should

$x.iter$	$y.iter$	(1)	(2)	(3)	(4)	$\text{cond}_{x.iter, y.iter}$
0	0	true	false	false	false	false
1	0	false	true	false	true	true
0	1	false	false	true	true	true
1	1	true	false	false	true	true
1	2	false	false	false	true	false

Figure 5.1: Truth table for  $\text{cond}_{x.iter, y.iter}$  with (1)  $x.iter = y.iter$ , (2)  $x.iter \neq 0 \wedge y.iter = 0$ , (3)  $x.iter = 0 \wedge y.iter \neq 0$  and (4)  $\neg(x.iter = 0 \wedge y.iter = 0)$ .

the results from both sides be equal, every node  $x$  should have a node  $y$  as its descendant and at the same time,  $x$  should itself be a descendant of  $y$ . This is not possible. Actually, every node  $x$  having a descendant  $y$  is itself an *ancestor* of that  $y$  node. To make the equation semantically correct, either one side of the equation should describe an **ancestor** axis step:

$$X \underset{\text{desc}::*}{\Downarrow} Y = Y \underset{\text{anc}::*}{\Downarrow} X \quad (5.5)$$

**Proof** The proof is simple. Using the symmetric staircase join definition of 5.3 equation 5.5 can be rewritten to:

$$\begin{aligned} & X \underset{\text{desc}::*}{\Downarrow} Y \\ &= \{ | (\langle \text{iter} = \max\{x.\text{iter}, y.\text{iter}\} \rangle \oplus (\langle_x (\tilde{\pi}_{\text{iter}}(x))) \oplus \\ &\quad (\langle_y (\tilde{\pi}_{\text{iter}}(y))) \mapsto 1) | \\ &\quad x \in X \wedge y \in Y \wedge y.\text{pre} > x.\text{pre} \wedge y.\text{post} < x.\text{post} \wedge \\ &\quad y.\text{kind} = \text{"element"} \wedge \text{cond}_{x.\text{iter}, y.\text{iter}} | \} \\ &= \{ | (\langle \text{iter} = \max\{y.\text{iter}, x.\text{iter}\} \rangle \oplus (\langle_y (\tilde{\pi}_{\text{iter}}(y))) \oplus \\ &\quad (\langle_x (\tilde{\pi}_{\text{iter}}(x))) \mapsto 1) | \\ &\quad y \in Y \wedge x \in X \wedge x.\text{pre} < y.\text{pre} \wedge x.\text{post} > y.\text{post} \wedge \\ &\quad x.\text{kind} = \text{"element"} \wedge \text{cond}_{y.\text{iter}, x.\text{iter}} | \} \\ &= Y \underset{\text{anc}::*}{\Downarrow} X \end{aligned}$$

□

A slight annotation has to be made however, on the applicability of this equivalence rule, depending on nodetest  $n$  (here:  $*$ ). This is illustrated by the following example.

**Example 5.5.** There is a relation  $X$  containing two elements:  $x_1$  of type *text* and  $x_2$  of type *node*. Additionally, there is a relation  $Y$ , also containing two elements:  $y_1$  of type *node*, and  $y_2$  of type *text*. The following relations exist:

- $x_1$  is an ancestor of  $y_1 \Leftrightarrow y_1$  is a descendant of  $x_1$
- $x_2$  is an ancestor of  $y_2 \Leftrightarrow y_2$  is a descendant of  $x_2$

Equation 5.5 holds:  $X \underset{\text{desc}::*}{\Downarrow} Y = Y \underset{\text{anc}::*}{\Downarrow} X$ . Both sides will evaluate to the result with node pairs  $(x_1, y_1)$  and  $(x_2, y_2)$ . Changing nodetest  $*$  to *text* results in:  $X \underset{\text{desc}::\text{text}}{\Downarrow} Y = Y \underset{\text{anc}::\text{text}}{\Downarrow} X$ . The left hand side will then evaluate to  $(x_2, y_2)$ , whereas the right hand side will yield  $(x_1, y_1)$ .

A similar example can be given for all other node types. The conclusion that can be drawn is that equation 5.5 is only universally applicable in combination with the arbitrary nodetest  $*$ . For other nodetests the applicability is context sensitive, i.e. depending on the operand data.  $\square$

There are similar equations to 5.5 for the left and right symmetric staircase join variants:

$$X \xrightarrow[\text{desc}::*]{\leftarrow} Y = Y \xrightarrow[\text{anc}::*]{\rightarrow} X \quad (5.6)$$

$$X \xrightarrow[\text{desc}::*]{\rightarrow} Y = Y \xrightarrow[\text{anc}::*]{\leftarrow} X \quad (5.7)$$

Their proofs are considered trivial and are therefore omitted.

All three equations 5.5 to 5.7 are constructed upon the XPath axes **ancestor** and **descendant**. These axes are the opposite of each other and are said to denote an *XPath symmetry*. Other pairs denoting symmetries are:

- **ancestor-or-self** - **descendant-or-self**
- **parent** - **child**
- **following** - **preceding**
- **following-sibling** - **preceding-sibling**
- **self** - **self** (trivial)

Equations 5.5 to 5.7 also hold for these axis pairs. More XPath symmetries are discussed in the next chapter.

## 5.4.2 Associativity

Apart from independent operands, the symmetric staircase join also provides the possibility to use its output as either a left or right operand of a subsequent staircase join operation. Associativity seems plausible. Suppose there are three relations  $X$ ,  $Y$ ,  $Z$  and axis  $\alpha$  is to be the **descendant** axis and  $n$  the arbitrary nodetest  $*$ . The equation for associativity now states that (see 5.4):

$$(X \xrightarrow[\text{desc}::*]{\leftarrow} Y) \xrightarrow[\text{desc}::*]{\leftarrow} Z = X \xrightarrow[\text{desc}::*]{\leftarrow} (Y \xrightarrow[\text{desc}::*]{\leftarrow} Z)$$

However, this expression is still syntactically incorrect. On the left hand side, first the descendants  $y \in Y$  are computed for every  $x \in X$ . Since this is done by means of a general staircase join variant, the result is passed to the next staircase join as a concatenation of tuples, each prefixed with either  $x$  or  $y$ . It is not possible to use this relation as a staircase join operand, it has to be transformed first, either through prefixing or projection of some sort.

This problem does not occur for the left and right staircase join variants. Therefore, the associativity equation will first be investigated for the right staircase join variant:

$$(X \xrightarrow[\text{desc}::*]{\vec{\sqcup}} Y) \xrightarrow[\text{desc}::*]{\vec{\sqcup}} Z = X \xrightarrow[\text{desc}::*]{\vec{\sqcup}} (Y \xrightarrow[\text{desc}::*]{\vec{\sqcup}} Z)$$

Now that the expression is syntactically correct, it is time to take a closer look at its semantics. First, the left hand side is written out using the definition of the symmetric staircase join:

$$\begin{aligned} & (X \xrightarrow[\text{desc}::*]{\vec{\sqcup}} Y) \xrightarrow[\text{desc}::*]{\vec{\sqcup}} Z \\ &= \{ | (\langle \text{iter} = \max\{t.\text{iter}, z.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(z) \mapsto 1) \mid t \in T \wedge z \in Z \wedge \\ & \quad T = \{ | (\langle \text{iter} = \max\{x.\text{iter}, y.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(y) \mapsto 1) \mid \\ & \quad \quad x \in X \wedge y \in Y \wedge y.\text{pre} > x.\text{pre} \wedge y.\text{post} < x.\text{post} \wedge \\ & \quad \quad y.\text{kind} = \text{"element"} \wedge \text{cond}_{x.\text{iter}, y.\text{iter}} \} \} \\ & \quad z.\text{pre} > t.\text{pre} \wedge z.\text{post} < t.\text{post} \wedge z.\text{kind} = \text{"element"} \wedge \\ & \quad \text{cond}_{t.\text{iter}, z.\text{iter}} \} \} \end{aligned}$$

The expression between parentheses is denoted by set  $T$ , which is nested in the set definition representing the second staircase join operation involving  $Z$ . Flattening the expression, the following set definition is obtained:

$$\begin{aligned} &= \{ | (\langle \text{iter} = \max\{x.\text{iter}, y.\text{iter}, z.\text{iter}\} \rangle \wedge \oplus \tilde{\pi}_{\text{iter}}(z) \mapsto 1) \mid \\ & \quad x \in X \wedge y \in Y \wedge z \in Z \wedge z.\text{pre} > y.\text{pre} > x.\text{pre} \wedge \\ & \quad z.\text{post} < y.\text{post} < x.\text{post} \wedge y.\text{kind} = \text{"element"} \wedge \\ & \quad z.\text{kind} = \text{"element"} \wedge \text{cond}_{x.\text{iter}, y.\text{iter}, z.\text{iter}} \} \} \end{aligned}$$

No further deduction is possible. The right hand side should now be written out. For the equation to hold, both sides should produce the same set.

$$\begin{aligned} & X \xrightarrow[\text{desc}::*]{\vec{\sqcup}} (Y \xrightarrow[\text{desc}::*]{\vec{\sqcup}} Z) \\ &= \{ | (\langle \text{iter} = \max\{x.\text{iter}, t.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(t) \mapsto 1) \mid t \in T \wedge \\ & \quad T = \{ | (\langle \text{iter} = \max\{y.\text{iter}, z.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(z) \mapsto 1) \mid \\ & \quad \quad y \in Y \wedge z \in Z \wedge z.\text{pre} > y.\text{pre} \wedge z.\text{post} < y.\text{post} \wedge \\ & \quad \quad z.\text{kind} = \text{"element"} \wedge \\ & \quad \quad \text{cond}_{y.\text{iter}, z.\text{iter}} \} \} \\ & \quad \wedge x \in X \wedge t.\text{pre} > x.\text{pre} \wedge t.\text{post} < x.\text{post} \wedge \\ & \quad t.\text{kind} = \text{"element"} \wedge \text{cond}_{x.\text{iter}, t.\text{iter}} \} \} \end{aligned}$$

Applying the same strategy and 'unnest'  $T$  yields:

$$= \{ | (\langle iter = \max\{x.iter, y.iter, z.iter\} \rangle \oplus \tilde{\pi}_{iter}(z) \mapsto 1) | \\ x \in X \wedge y \in Y \wedge z \in Z \wedge z.pre > x.pre \wedge z.pre > y.pre \wedge \\ z.post < x.post \wedge z.post < y.post \wedge y.kind = "element" \wedge \\ z.kind = "element" \wedge cond_{x.iter, y.iter, z.iter} | \}$$

Without further specifying the conditions  $cond_{x.iter, y.iter, z.iter}$ , but solely on the  $(pre, post)$  it can be seen that the left and right hand sides of the equation evaluate to two sets that do not have to be equal. On the left hand side, there is are explicit relations defined between  $x$  and  $y$ :

- $y.pre > x.pre$
- $y.post < x.post$

These relations are not defined in the result of the right hand side. This is due to the direction of the staircase join. The right staircase join on the left hand side first determines all  $y$  descendants for nodes  $x$ , before determining the  $z$  descendants of  $y$ . Here,  $x$ ,  $y$  and  $z$  define different elements. On the right hand side however, the  $z$  descendants of each  $y$  node are determined first. Subsequently, all  $z$  nodes are produced that are descendants of an  $x$  node. This does not exclude the case where an  $x$  node is also a  $y$  node.

By analogous proof, the same statement can be made for the left symmetric staircase join. Rounding up, neither of the three symmetric staircase join variants support associativity:

- $(X \xrightarrow[\text{desc}::*]{\leftarrow} Y) \xrightarrow[\text{desc}::*]{\leftarrow} Z \neq X \xrightarrow[\text{desc}::*]{\leftarrow} (Y \xrightarrow[\text{desc}::*]{\leftarrow} Z)$
- $(X \xrightarrow[\text{desc}::*]{\rightarrow} Y) \xrightarrow[\text{desc}::*]{\rightarrow} Z \neq X \xrightarrow[\text{desc}::*]{\rightarrow} (Y \xrightarrow[\text{desc}::*]{\rightarrow} Z)$
- $(X \xleftarrow[\text{desc}::*]{\leftarrow} Y) \xleftarrow[\text{desc}::*]{\leftarrow} Z \neq X \xleftarrow[\text{desc}::*]{\leftarrow} (Y \xleftarrow[\text{desc}::*]{\leftarrow} Z)$

There are however two syntactically correct variations that have not been discussed yet. These two equations involve combinations of a left and a right symmetric staircase join variant. First, consider the equation:

$$(X \xleftarrow[\text{desc}::*]{\leftarrow} Y) \xrightarrow[\text{desc}::*]{\rightarrow} Z = X \xleftarrow[\text{desc}::*]{\leftarrow} (Y \xrightarrow[\text{desc}::*]{\rightarrow} Z)$$

Semantically, this equation is still not correct. The left hand side first determines all  $x$  nodes that have a  $y$  node as their descendant. Subsequently, all descendants  $z$  of these  $x$  nodes are computed. No statement is made about the relation between the  $y$  and  $z$  descendants of  $x$ . Either  $y$  is a descendant of  $z$  or vice versa. The right hand side however, first determines all  $z$  descendants of  $y$ . Subsequently, all  $x$  nodes are computed that have such a  $z$

node as their descendant. Here, it is made explicit that nodes  $z$  should be descendants of  $y$  nodes. It is even possible for the  $y$  nodes to be the *ancestor* of a node  $x$ . It is obvious that both sides do not have to produce the same result.

The equation presented here however, is both syntactically and semantically correct:

$$(X \xrightarrow[\text{desc}::*]{\rceil} Y) \xleftarrow[\text{desc}::*]{\lceil} Z = X \xrightarrow[\text{desc}::*]{\rceil} (Y \xleftarrow[\text{desc}::*]{\lceil} Z) \quad (5.8)$$

**Proof** First, the left hand side is written out using the left and right symmetric staircase join definitions:

$$\begin{aligned} & (X \xrightarrow[\text{desc}::*]{\rceil} Y) \xleftarrow[\text{desc}::*]{\lceil} Z \\ &= \{ | (\langle \text{iter} = \max\{t.\text{iter}, z.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(t) \mapsto 1) \mid t \in T \wedge \\ & \quad T = \{ | (\langle \text{iter} = \max\{x.\text{iter}, y.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(y) \mapsto 1) \mid \\ & \quad \quad x \in X \wedge y \in Y \wedge y.\text{pre} > x.\text{pre} \wedge y.\text{post} < x.\text{post} \wedge \\ & \quad \quad y.\text{kind} = \text{"element"} \wedge \\ & \quad \quad \text{cond}_{x.\text{iter}, y.\text{iter}} \} \} \\ & \quad \wedge z \in Z \wedge z.\text{pre} > t.\text{pre} \wedge z.\text{post} < t.\text{post} \wedge z.\text{kind} = \text{"element"} \wedge \\ & \quad \text{cond}_{t.\text{iter}, z.\text{iter}} \} \} \end{aligned}$$

As done before, this set expression is rewritten to yield:

$$\begin{aligned} &= \{ | (\langle \text{iter} = \max\{x.\text{iter}, y.\text{iter}, z.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(y) \mapsto 1) \mid \\ & \quad x \in X \wedge y \in Y \wedge z \in Z \wedge z.\text{pre} > y.\text{pre} > x.\text{pre} \wedge \\ & \quad z.\text{post} < y.\text{post} < x.\text{post} \wedge y.\text{kind} = \text{"element"} \wedge \\ & \quad z.\text{kind} = \text{"element"} \wedge \text{cond}_{x.\text{iter}, y.\text{iter}, z.\text{iter}} \} \} \end{aligned}$$

The same procedure is applied to the right hand side of the equation:

$$\begin{aligned} & X \xrightarrow[\text{desc}::*]{\rceil} (Y \xleftarrow[\text{desc}::*]{\lceil} Z) \\ &= \{ | (\langle \text{iter} = \max\{x.\text{iter}, t.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(t) \mapsto 1) \mid t \in T \wedge \\ & \quad T = \{ | (\langle \text{iter} = \max\{y.\text{iter}, z.\text{iter}\} \rangle \oplus \tilde{\pi}_{\text{iter}}(y) \mapsto 1) \mid \\ & \quad \quad y \in Y \wedge z \in Z \wedge z.\text{pre} > y.\text{pre} \wedge z.\text{post} < y.\text{post} \wedge \\ & \quad \quad z.\text{kind} = \text{"element"} \wedge \\ & \quad \quad \text{cond}_{y.\text{iter}, z.\text{iter}} \} \} \\ & \quad \wedge x \in X \wedge t.\text{pre} > x.\text{pre} \wedge t.\text{post} < x.\text{post} \wedge \\ & \quad t.\text{kind} = \text{"element"} \wedge \\ & \quad \text{cond}_{x.\text{iter}, t.\text{iter}} \} \} \end{aligned}$$

$$= \{ | (\langle iter = \max\{x.iter, y.iter, z.iter\} \rangle \oplus \tilde{\pi}_{iter}(y) \mapsto 1) | \\ x \in X \wedge y \in Y \wedge z \in Z \wedge z.pre > y.pre > x.pre \wedge \\ z.post < y.post < x.post \wedge y.kind = \text{"element"} \wedge \\ z.kind = \text{"element"} \wedge cond_{x.iter, y.iter, z.iter} | \}$$

Both sides of the equation deliver a result set based on the same (*pre*, *post*) conditions. It can be assumed that the equation is valid. However, to be able to make this statement, the conditions  $cond_{x.iter, y.iter, z.iter}$  are to be further elaborated upon.

For two relations  $X$  and  $Y$  with elements  $x$  and  $y$  respectively, the symmetric staircase join operation  $X \stackrel{\alpha::n}{\sqcup} Y$  is only defined for the tree cases where either:

- $x.iter = y.iter \neq 0$
- $x.iter = 0 \wedge y.iter \neq 0$
- $x.iter \neq 0 \wedge y.iter = 0$

These three cases have already been expressed by the conditions depicted in definition 5.4. Now consider the expressions of equation 5.8. Both expressions involve two staircase join operations and three relations  $X$ ,  $Y$  and  $Z$ . For an expression to be valid, the conditions on the *iter* attribute of elements  $x$ ,  $y$  and  $z$  have to be constructed such that neither of the two staircase join operators will possibly face two operands with *iter* attributes equal to zero. The expression on the left hand side is valid for:

- $x.iter = y.iter = z.iter \neq 0$
- $x.iter \neq 0 \wedge y.iter = 0 \wedge z.iter \neq 0$
- $x.iter = 0 \wedge y.iter \neq 0 \wedge z.iter = 0$
- $x.iter \neq 0 \wedge y.iter = 0 \wedge z.iter = 0$

For the right hand side, the following cases can be distinguished:

- $x.iter = y.iter = z.iter \neq 0$
- $x.iter \neq 0 \wedge y.iter = 0 \wedge z.iter \neq 0$
- $x.iter = 0 \wedge y.iter \neq 0 \wedge z.iter = 0$
- $x.iter \neq 0 \wedge y.iter = 0 \wedge z.iter = 0$

On both sides, the first three cases correspond to each other. The fourth expression on both sides however, will not result in a valid expression on the other side of the equation. Therefore, the obtained equivalence rule of equation 5.8 is only valid for the *iter* combinations depicted in figure 5.2.

□

$x.iter$	$y.iter$	$z.iter$
$n$	$n$	$n$
$\neq 0$	$0$	$\neq 0$
$0$	$\neq 0$	$0$

Figure 5.2: Combinations of *iter* for which equation 5.8 is valid,  $n \in \{1, 2, 3, \dots\}$

## 5.5 Pushing operators through a staircase join

In addition to better support for equivalence rules as shown above, the symmetric staircase join provides better possibilities for pushing other operators through the staircase join than the asymmetric staircase join. This will be illustrated by an example.

**Example 5.6.** Consider the expression:

$$\sigma_p(C \underset{\text{desc}::*}{\bowtie} D)$$

For context nodes  $c \in C$ , their descendants  $d \in D$  are computed. On the descendants, a selection (with condition  $p$ ) is performed to obtain the result of the expression. To improve efficiency, one might push the selection through the staircase join:

$$C \underset{\text{desc}::*}{\bowtie} (\sigma_p(D))$$

**Asymmetric staircase join** Assume that the staircase join used is asymmetric. Then, relation  $C$  is dependent of  $D$ : during evaluation all properties of  $c \in C$  have to be looked up in  $D$ . Suppose that some node  $c \in C$  does not conform to the condition depicted by  $p$ :  $\sigma_p$  will then 'remove' all properties of node  $c$  from  $D$ . The lookup of its properties will fail and the staircase join is now unable to determine its descendants.

**Symmetric staircase join** Had the staircase join been symmetric, there should not have occurred any problem. Since both operands of the symmetric staircase join are defined on the same domain, providing all necessary properties, no additional lookups are required to compute the result. The non-existence of the properties of node  $c$  in  $D$  does not prevent the symmetric staircase join of obtaining its descendants.

□

This example also applies to the left and right staircase join variants. Pushing an operator to the left operand does not affect either the asymmetric or the symmetric staircase join.

## Chapter 6

# Xpath symmetries

In the previous chapter, the subject of XPath symmetries has already been briefly touched. This chapter explores these symmetries in more detail guided by the article [11] on rewriting XPath expressions. In this article, the *rare* algorithm is presented: **reverse axis removal**. By converting all reverse axis steps in an XPath expression to forward axis steps, *rare* makes it possible to query large documents (> main memory size) stream based in one sequential pass. XPath symmetries are defined by the symmetric axis pairs, as depicted in figure 6.1 <sup>1</sup>.

As [11] states, all *reverse* axis steps can be transformed into *forward* axis steps. For every reverse step, there exists an equivalent only using forward axes. Also, in the opposite direction, there exists an equivalent reverse XPath expression for every forward XPath expression. Furthermore, expressions combining both reverse and forward axes may be rewritten in either reverse or forward expressions. This chapter explores the possibilities of recognizing XPath symmetries within an algebra DAG and rewriting them to their symmetric equivalent.

### 6.1 Location path equivalence

To be able to tell anything about symmetric equivalence, the concept of equivalence of XPath location paths has first to be defined.

**Definition 6.1.** An XPath location path  $p$  is defined as an arbitrary number of XPath axis steps, possibly containing (nested) predicates, which on their turn may contain a location path.  $\square$

**Definition 6.2.** Let  $p_1$  and  $p_2$  be two XPath location paths,  $C$  the set of context nodes in which  $p_1$  and  $p_2$  originate and  $R$  the result set. Then, the

---

<sup>1</sup>Note that the pair (`self,self`) is trivial, but a useful symmetry in rewriting XPath expressions nonetheless.

two location paths  $p_1$  and  $p_2$  are equivalent if they yield the same result  $R$  after evaluation originating in context  $C$ .  $\square$

Predicates may be arbitrarily nested. Nested predicates however only affect the complexity of the path expression itself, they do not affect the main principle of XPath symmetries. Therefore, nested predicates are omitted in the remaining of this thesis.

forward	reverse
child	parent
descendant	ancestor
descendant-or-self	ancestor-or-self
following	preceding
following-sibling	preceding-sibling
self	self

Figure 6.1: Symmetrical XPath axis pairs.

## 6.2 A simple XPath symmetry

This chapter uses the same *auction.xml* as before. Assume a query is needed to perform the following:

*Give a list of all persons that have a name defined.*

This could be solved by a simple XPath expression:

$$/descendant::name/parent::person$$

The approach is to take each *name* descending from the root and return its parent if it is a *person*. Two axis steps are performed, a forward axis step **descendant** and a reverse axis step **parent**. To eliminate the **parent** step and only use *forward* steps, a different approach is needed.

A possible approach would be to take all *person* nodes (note that the expression above does not exclude the root from being a person) and check whether they have a *name* child. Expressing this in XPath yields:

$$/descendant-or-self::person[./child::name]$$

It is obvious that both expressions define the same set of person elements and therefore:

$$\begin{aligned} /descendant::name/parent::person = \\ /descendant-or-self::person[./child::name] \end{aligned} \tag{6.1}$$

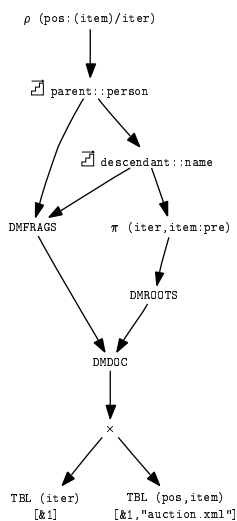


Figure 6.2: DAG corresponding to the left hand side of equation 6.1

### 6.2.1 Compiling the equation

Using the same steps as in chapter 3 section 3.4 for both sides of equation 6.1 the corresponding DAG can be constructed.

**Left hand side with reverse axis** Compiling the left hand side to XQuery Core is very straightforward and simple. Two XPath steps are performed and no for-loop is needed:

```

XPATH (XPATH (XFNDOC (XSTR "auction.xml"))
        (Descendant, XMLElem, ["name"]))
      (Parent, XMLElem, ["person"])
  
```

Using the prototype of the Pathfinder compiler, the XQuery Core expression leads to the DAG depicted in figure 6.2<sup>2</sup>.

**Right hand side without reverse axes** Constructing the symmetrical equivalent on the right hand side of equation 6.1 has introduced a predicate expression. Compiling expressions with predicates to XQuery Core is slightly more difficult than compiling expressions without predicates. A predicate is considered a conditional expression. Here, *persons* are selected based on if they have a *name* child or not. This check is embedded in a for-loop, as it has to be evaluated for all *person* elements in the document.

<sup>2</sup>Note that it is made explicit that the staircase join currently implemented in Pathfinder is in fact a *right* staircase join.

Expressed in XQuery Core, this leads to:

```

XFOR "b" (XPATH (XFNDOC (XSTR "auction.xml"))
            (Descendant_or_self, XMLElem, ["person"]))
(XIF (XFNNOT (XFEMPTY (XPATH (XVAR "b")
                             (Child, XMLElem, ["name"]))))
      (XVAR "b")
      XEMPTY
    )
  )

```

Compiling this expression results in the DAG of figure 6.3.

### 6.2.2 DAG Comparison

When comparing the DAGs of figures 6.2 and 6.3, the first thing that jumps to attention is the major difference in size. While the DAG corresponding to the left hand side of equation 6.1 is constructed by 10 nodes only, the DAG of the right hand side has 20, a difference of a factor 2!

In the DAG of figure 6.3 it can be clearly seen that loop lifting is performed. How to recognize loop lifting in a DAG has already been explained in chapter 3. The loop adds two more joins to the two staircase joins that were already present in the DAG started with (figure 6.2). In this loop, the axis step and conditional expression of the `child::name` predicate are performed. Figure 6.4 lists all these differences.

## 6.3 Pattern recognition

The question is: given an arbitrary DAG, could XPath symmetries be recognized and rewritten into their symmetric equivalent? To establish this, there should be a pattern where XPath symmetries comply to. The DAG of figure 6.2 follows a pattern that is easy to recognize. A concatenation of axis steps (staircase joins) defines a simple XPath expression without predicates, for which a symmetrical equivalent can be easily determined.

For the DAG of figure 6.3 the situation is much more complicated. As was the case in figure 6.2 there are two staircase join operations, only now, one of them is embedded in a for-loop, together with an additional condition. This for-loop, the embedded axis step and the condition form the `[/child::name]` predicate of the XPath expression. However, not every for-loop containing a staircase join operation has to denote a predicate, see example 3.4 in chapter 3. A clear pattern cannot be determined.

## 6.4 Left staircase join

As shown above, pattern matching predicates in a DAG is very complicated. To perform an axis step and then traverse back requires a loop and possi-

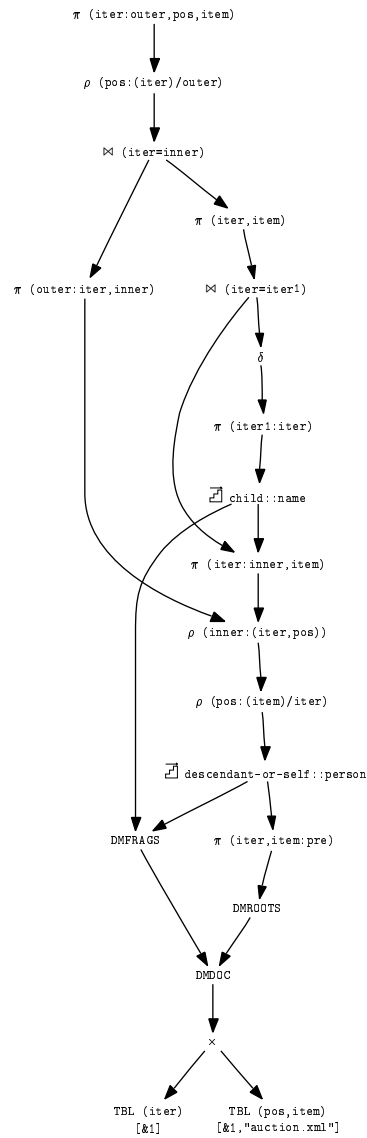
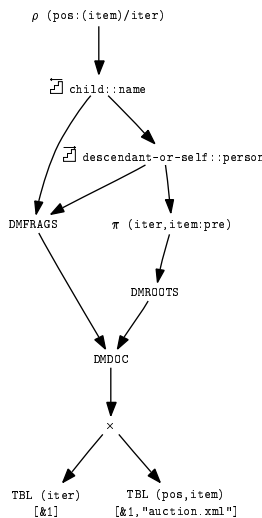


Figure 6.3: DAG corresponding to the right hand side of equation 6.1

DAG of figure 6.2	DAG of figure 6.3
2 staircase joins	2 staircase joins
no additional joins	2 equi-joins
no looplifting	loop lifting
no conditional expression	conditional expression

Figure 6.4: Comparing the two DAGs

Figure 6.5: DAG introducing the *left* staircase join.

bly some additional logic. However, chapter 5 introduced some interesting staircase join variants. Among these the *left* staircase join, which is defined to return the context nodes of the XPath axis step it performs. This is exactly what happens in the predicate of 6.1. Changing the predicate to a left staircase join reduces the DAG of figure 6.3 to the DAG depicted in figure 6.5.

The left staircase join has reduced the DAG of 6.3 from 20 to only 10 graph nodes, which is the same size as its symmetrical equivalent of 6.2. This constitutes the need for a left staircase join implementation for evaluating predicates.

## 6.5 Symmetric staircase join

As opposed to the asymmetric staircase join so far considered in this chapter, the symmetric staircase join provides even more interesting rewriting possibilities. It does not require any of its operands to be the document relation. An DAG equivalent to the DAG of figure 6.3 using the symmetric staircase join is shown in figure 6.6. First, two staircase joins are performed to construct two relations. One relation contains all *person* elements, the other will contain all *name* elements of the document. These two relations are used as input for a third staircase join operation, which will determine which persons have a name among their children.

Although there are now three staircase join operations involved as opposed to two in the query plan of figure 6.5, the query plan based on the symmetric staircase join might be more efficient. This of course depends on

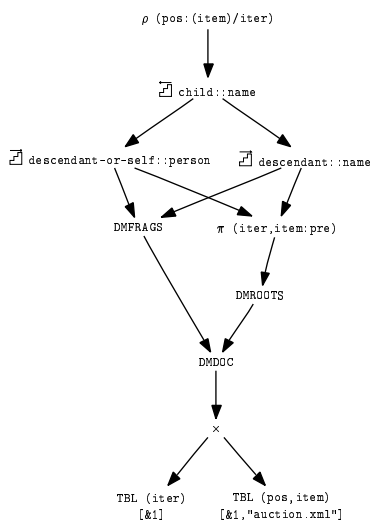


Figure 6.6: DAG introducing the *symmetric* staircase join.

the document structure. For example, the plan of 6.6 might not be such a good idea if there are many other (not person) elements with a *name* child.

## 6.6 Complex XPath expressions

Since the main conclusion on recognizing and rewriting XPath expressions is already made, there is no need to further elaborate on more complex expressions. In the current Pathfinder implementation, a simple location path can be rewritten to a very large and complicated symmetric equivalent. The complexity of this symmetric equivalent might be greatly reduced by implementing the left staircase join variant. Furthermore, the general staircase join should also be considered for this purpose, when even more complicated XPath expressions occur. This has already been mentioned in chapter 5.

A different approach to XPath location path rewriting would be to recognize symmetries and rewrite them on a higher level. What is a cumbersome task within the algebra might be less complicated in XQuery Core or XQuery itself. [11] already provides the rare algorithm that serves this purpose.

# Chapter 7

## Conclusion

This thesis has discussed the possibilities of optimizing XPath expressions on the level of the relational algebra of the Pathfinder XQuery system. Before drawing the final conclusions, the next section gives a brief summary of the research lining out the achievements made. Based on these achievements the final conclusions are made, followed by a section presenting areas for future research.

### 7.1 Summary

**Algebra specification** The principal of optimization is to rewrite some expression into another expression that produces the same result. Equal expressions define equivalence rules. To formally prove the validity of such equivalence rules, the entire relational model of Pathfinder has been specified. Relational tables are considered to be multi-sets of records. Subsequently, using set theory, each operator has also be defined as a similar multi-set of records.

**Staircase join** The current implementation of Pathfinder hosts an implementation of the *right* variant of the staircase join. As has been shown in this thesis, this implementation is very limited in its optimization possibilities. To improve flexibility in this matter, two additional variants of the staircase join have been proposed:

- left staircase join
- general staircase join

Both these variants are designed to efficiently handle simple predicates (left) and more complicated predicates (general) in XPath expressions. Still, all three variants are not flexible in operand order and order of evaluation: they all fail for commutativity and associativity. Reason is that in essence,

the current implementation of the staircase join is a selection rather than a join operator. The definition of the symmetric staircase join supports all properties of the current staircase join implementation and adds some interesting rewriting possibilities. The symmetric staircase join can also be made available in a left and right variant.

Left and right staircase join variants can be expressed in terms of a general staircase join as (with  $C$  a context relation,  $D$  the document relation,  $\alpha$  some axis step and  $n$  a nodetest):

- $C \xrightarrow[\alpha::n]{\text{⌋}} D = \triangleright_d(\tilde{\pi}_{c.*}(C \text{⌋}_{\alpha::n} D))$
- $C \xleftarrow[\alpha::n]{\text{⌋}} D = \triangleright_c(\tilde{\pi}_{d.*}(C \text{⌋}_{\alpha::n} D))$

Neither of the three staircase join variants support commutativity or associativity ( $C_1$  and  $C_2$  are both context relations):

- $C \text{⌋}_{\alpha::n} D \neq D \text{⌋}_{\alpha::n} C$
- $(C_1 \text{⌋}_{\alpha::n} C_2) \text{⌋}_{\alpha::n} D \neq C_1 \text{⌋}_{\alpha::n} (C_2 \text{⌋}_{\alpha::n} D)$

Equivalence rules that have been found for the symmetric staircase join which resemble the equations for commutativity and associativity above, on operands  $X$ ,  $Y$  and  $Z$ , are:

- $X \text{⌋}_{f::*} Y = Y \text{⌋}_{r::*} X$
- $X \xleftarrow[\alpha::*]{\text{⌋}} Y = Y \xrightarrow[\alpha::*]{\text{⌋}} X$
- $X \xrightarrow[\alpha::*]{\text{⌋}} Y = Y \xleftarrow[\alpha::*]{\text{⌋}} X$
- $(X \xrightarrow[\alpha::*]{\text{⌋}} Y) \xleftarrow[\alpha::*]{\text{⌋}} Z = X \xrightarrow[\alpha::*]{\text{⌋}} (Y \xleftarrow[\alpha::*]{\text{⌋}} Z)$

In the first three equations, the axis pair  $(f, r)$  resembles one of the (*forward*, *reverse*) symmetrical pairs of:

forward	reverse
child	parent
descendant	ancestor
descendant-or-self	ancestor-or-self
following	preceding
following-sibling	preceding-sibling
self	self

The fourth equation is only valid for the following two combinations of *iter* values ( $n \in \{1, 2, 3, \dots\}$ ):

$x.iter$	$y.iter$	$z.iter$
$n$	$n$	$n$
$\neq 0$	$0$	$\neq 0$
$0$	$\neq 0$	$0$

**XPath symmetries** In [11] it has been shown how XPath expressions containing reverse axes can be rewritten into their symmetric equivalent only containing forward axis steps. Example XPath symmetries of this article have been projected onto Pathfinder’s relational algebra. Comparing the corresponding query plans, it has been shown that they all follow the same pattern. When rewriting forward path expressions to reverse expressions and vice versa, one of the expressions always has to contain one or more predicates. The algebraic pattern for a predicate consists of:

- loop lifting
- one or more axis steps (staircase joins) inside the loop
- possible additional conditions

In contrast, a path expression without predicates will only contain a sequence of staircase join operations. Although complex path expressions will always comply to the structure of these patterns, the expression containing one or more predicates might be very complicated. Therefore, it will be very difficult to recognize the pattern.

**In practise** To verify the statements made in this thesis and for generating query plans, a Haskell prototype of the Pathfinder XQuery compiler has been used. Apart from using the prototype for testing purposes, it has also been extended for better XPath support. Absent axes were added, except for the axes XPathnamespace and XPathattribute. Namespaces have been omitted in this research. As all attributes have been treated as elements, the specific axis was not needed. The left staircase join has been implemented for predicate evaluation. Finally, the prototype is extended with an optimizer module that performs some small optimizations by removing superfluous operations and attributes.

## 7.2 Conclusions

**Staircase join** The staircase join in its currently implemented form has proven to be very limited in its rewriting possibilities. Adding the variants for a *left*, *general* and a symmetric version of the staircase join improves

this. Making the staircase join symmetric causes it to behave more like a regular join operator. Both operands of the symmetric staircase join are defined on same domain (same set of attributes). This provides better ground for pushing operands such as selection and projection to either of the two operands.

**XPath symmetries** XPath symmetries have proven to be a very interesting area with a lot of optimization possibilities. Although equivalent path expressions do comply to a certain pattern, they are very difficult to identify. Recognizing the pattern will be a complicated and cumbersome task for large path expressions containing predicates. It is expected that XPath symmetries are much easier to recognize on the level of XQuery Core. In [11], it has already been proven that reverse path expressions can be algorithmically rewritten into their forward equivalent.

**Optimization possibilities** Rewriting query plans is a basis for query optimization. This thesis has shown how query plans generated by Pathfinder can be rewritten in different forms. However, based only on query plan knowledge, it is very difficult to predict whether a rewritten version of a query plan is more efficient than its original. A cost-based analysis has to be made in order to give a solid ground for choosing the best alternative. In this analysis, various aspects have to be considered, among which:

- schema information
- intermediate result size
- relational properties (e.g. key information, dense numbering, etc.)
- query plan size

Performing such a full analysis could be very time consuming and therefore possibly not desirable for simple queries as it might even exceed the actual query time. Query optimization is a very difficult and delicate matter as compromises have to be made all the time.

### 7.3 Future research

**Higher level optimization** This thesis has described certain optimization possibilities on the algebraic level. However, on the algebraic level, not all optimizations could be (efficiently) performed. As these optimizations might yield some very interesting results (for example recognizing and rewriting XPath symmetries), they should be re-investigated on a higher level.

**Staircase join variants** It has been shown that the staircase join in its currently implemented form lacks functionality to fully exploit the optimization possibilities. This can be improved by making the staircase join symmetric. For efficient handling of path expression predicates, the left staircase join is highly suitable. It is advisable to implement these two variants of the staircase join.

**Schema information** In Pathfinder, it is not required to supply an XML Schema or DTD along with an XML document. Without such a schema, as has been the case considered in this thesis, it is not possible to exploit knowledge about the structure of the XML tree. Schema's might supply important info on the element relationships and the level of nesting, which might be very useful in query rewriting.

**Multiple document fragments** So far, only single document queries have been investigated. The database however is capable of storing more than one XML document or even storing document fragments that together form one large document. Multiple document queries might be optimized by comparing fragment sizes or by splitting up the query to the different fragments based on available schema information.

# Bibliography

- [1] M. Fernández e.a., XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft, World Wide Web Consortium, July 2004, <http://www.w3.org/TR/xpath-datamodel/>.
- [2] MonetDB, © CWI 1994-2005, <http://monetdb.cwi.nl/>.
- [3] T. Grust. Accelerating XPath Location Steps. In *Proc. of the 21st International ACM SIGMOD Conference on Management of Data*, pp 109-120, Madison, Wisconsin, USA, June 2002.
- [4] T. Grust, M. van Keulen, J. Teubner, Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of the 28th VLDB Conference*, Berlin, Germany, 2003.
- [5] T. Grust, J. Teubner, Relational Algebra: Mother Tongue–XQuery: Fluent. TDM’04, the first Twente Data Management Workshop on XML Databases and Information Retrieval, Enschede, The Netherlands, 2004.
- [6] D. Draper e.a., XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, World Wide Web Consortium, February 2004, <http://www.w3.org/TR/xquery-semantics/>.
- [7] Haskell, A Purely Functional Language, <http://www.haskell.org/>.
- [8] M. van Keulen, Relational approach to logical query optimization of XPath. In *Proceedings of the 1st Twente Data Management Workshop (TDM’04) on XML Databases and Information Retrieval*, pp 52-58, Enschede, The Netherlands, June 2004.
- [9] XMark - An XML Benchmark, © CWI 2002-2003, <http://www.xml-benchmark.org/>.
- [10] P. Boncz, T.Grust, S. Manegold, J. Rittinger, J. Teubner, Pathfinder: Relational XQuery Over Multi-Gigabyte XML Inputs In Interactive Time. In *Proceedings of the 31st VLDB Conference*, Trondheim, Norway, 2005.

- [11] D. Olteanu, H. Meuss, T. Furche, F. Bry, XPath: Looking Forward. In *Proceedings of Workshop on XML-Based Data Management at EDBT*, Prague, Czech Republic, 2002.
- [12] M. Brantner, S. Helmer, C. Kanne, G. Moekotte, Full-fledged Algebraic XPath Processing in Natix. In *Proceedings of the 21st International Conference on Data Engineering*, Tokyo 2005.

## Appendix A

# DTD for auction.xml

```
<!-- 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)>

```

## Appendix B

# XQuery Core dialect

The Haskell prototype of Pathfinder does not include an XQuery Core parser. Instead, XQuery Core expressions are defined using the Core data type. The supported dialect and how it is can be used to define Core expressions:

```
data Core = XINT      Integer
          | XSTR      String          -- "foo"
          | XDBL      Double         -- 4.2e1
          | XDEC      Float          -- 4.20
          | XEMPTY    ()              -- ()
          | XVAR      QName          -- $x
          | XPLUS     Core Core      -- e1 + e2
          | XTIMES    Core Core      -- e1 * e2
          | XMINUS    Core Core      -- e1 - e2
          | XDIV      Core Core      -- e1 div e2
          | XIDIV     Core Core      -- e1 idiv e2
          | XMOD      Core Core      -- e1 mod e2
          | XLT       Core Core      -- e1 < e2
          | XGT       Core Core      -- e1 > e2
          | XEQ       Core Core      -- e1 = e2
          | XOR       Core Core      -- e1 and e2
          | XAND      Core Core      -- e1 or e2
          | XNEG      Core           -- - e
          | XIS       Core Core      -- e1 is e2
          | XBEFORE   Core Core      -- e1 << e2
          | XAFTER    Core Core      -- e1 >> e2
          | XFNNOT    Core           -- fn:not (e)
          | XFNSUM    Core           -- fn:sum (e)
          | XFNCOUNT  Core           -- fn:count (e)
          | XFNEMPTY  Core           -- fn:empty (e)
          | XFNDIST   Core           -- fn:distinct-values (e)
```

XFNDDO	Core	-- fn:distinct-doc-order (e)
XFNDOC	Core	-- fn:doc (e)
XFNDATA	Core	-- fn:data (e)
XFNTRUE		-- fn:true ()
XFNFALSE		-- fn:false ()
XFNROOT	Core	-- fn:root (e)
XCASTINT	Core	-- xs:integer (e)
XCASTSTR	Core	-- xs:string (e)
XCASTDEC	Core	-- xs:decimal (e)
XCASTDBL	Core	-- xs:double (e)
XTYPESW	Core SeqTy Core Core	-- typeswitch (e1) -- case t return e2 -- default return e3
XLET	QName Core Core	-- let \$x := e1 return e2
XFOR	QName Core Core	-- for \$x in e1 return e2
XFORAT	QName QName Core Core	-- for \$x at \$p in e1 return e2
XORDER	Core [Core]	-- e1 order by e2, e3, ..., en
XIF	Core Core Core	-- if e1 then e2 else e3
XSEQ	Core Core	-- e1, e2
XFUN	QName [Core]	-- f (e1, e2, ..., en)
XELEM	Core Core	-- element {e1} {e2}
XTEXT	Core	-- text {e}
XPATH	Core XPstep	-- e / ax::n