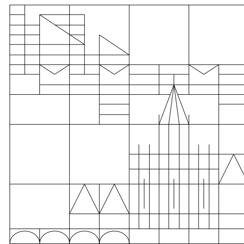
Master Thesis

Pathfinder/MonetDB:
A Relational Runtime for XQuery

Jan Rittinger

jan.rittinger@uni-konstanz.de



University of Konstanz, Germany

August 2005

Pathfinder/MonetDB: A Relational Runtime for XQuery
Master Thesis*, August 2005

Author: Jan Rittinger

First assessor:
Prof. Dr. Torsten Grust
Fakultät für Informatik, Lehrstuhl III: Datenbanksysteme
Technische Universität München
`http://www-db.in.tum.de/`

Second assessor:
Prof. Dr. Marc H. Scholl
Database & Information Systems Group
University of Konstanz
`http://www.inf.uni-konstanz.de/dbis/`

Urhebervermerk Ich versichere, dass ich die vorliegende Arbeit selbstständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

---

*) This work was supported by the DFG Research Training Group GK-1042 ”Explorative Analysis and Visualization of large Information Spaces”.

Contents

1 Introduction **1**
1.1 Outline . . . 1
1.2 The XML Query Language: XQuery . . . 1
1.3 MonetDB . . . 3
1.4 Pathfinder . . . 5

2 Relational Storage **6**
2.1 pre/post Encoding . . . 6
2.2 XML Document Storage . . . 7
2.3 Item Sequence Encoding . . . 10
2.4 Item Storage . . . 10

3 Core to MIL Translation **11**
3.1 Loop-Lifting . . . 11
3.2 Relational Operators . . . 13
3.3 Inference Rule Notation . . . 14
3.4 Algebraic Translation Rules . . . 15
3.4.1 Constants . . . 16
3.4.2 Sequences . . . 16
3.4.3 Variable References . . . 17
3.4.4 Let Bindings . . . 17
3.4.5 For Expressions . . . 18
3.4.6 If-Then-Else . . . 19
3.4.7 Typeswitch . . . 20
3.4.8 Path Steps . . . 20
3.4.9 Text Constructor . . . 21
3.4.10 Attribute Constructor . . . 22
3.4.11 Element Constructor . . . 23
3.4.12 Count . . . 28
3.4.13 Arithmetic and Comparison Operators . . . 28
3.4.14 Order by Expression . . . 29
3.4.15 Functions . . . 31
3.4.16 Casts . . . 31
3.5 Translation to MIL . . . 31
3.5.1 Constants . . . 32
3.5.2 Sequences . . . 33
3.5.3 Let Bindings and Variable References . . . 34
3.5.4 For Expressions . . . 34
3.5.5 If-Then-Else . . . 35
3.5.6 Typeswitch . . . 35
3.5.7 Path Steps and Constructors . . . 36
3.5.8 Count . . . 37

3.5.9 Arithmetic and Comparison Operators . . . 37
3.5.10 Order by Expressions . . . 37
3.5.11 Built-In Functions . . . 38
3.5.12 User-Defined Functions . . . 38

4 Optimizations **39**
4.1 Order Awareness . . . 39
4.1.1 Merged Union . . . 40
4.1.2 Conclusion . . . 41
4.2 Loop-Lifted Path Steps . . . 41
4.2.1 Basic Staircase Join . . . 42
4.2.2 Loop-Lifted Child Step . . . 47
4.2.3 Loop-Lifted Descendant Step . . . 47
4.2.4 Early Name and Kind Tests . . . 50
4.2.5 Loop-Lifted Path Steps for Other Axes . . . 52
4.3 Avoiding the iter|pos|item|kind Interface . . . 54
4.4 Join Recognition . . . 55
4.4.1 Join Pattern . . . 56
4.4.2 Join Translation . . . 57
4.4.3 Existential Semantics . . . 60

5 Experiments **63**
5.1 Order Awareness . . . 63
5.2 Loop-Lifted Path Steps . . . 64
5.3 Interfaces . . . 65
5.4 Join Recognition . . . 65
5.5 Scaling . . . 66
5.6 System Comparison . . . 67

6 Conclusions and Outlook **69**

Bibliography **72**

Acknowledgments **74**

A XMark Queries **75**

Abstract

This master thesis proposes the use of a relational database as special query processor for the XML query language XQuery. We chose MonetDB, an extensible RDBMS, to become our relational back-end. Its low level interpreter language MIL, which combines a relational algebra and a procedural language, became our target language for the XQuery compilation. The thesis first sketches concepts of the two languages as well as general ideas of the MonetDB DBMS and the Pathfinder compiler. The overview is followed by the description of storage structures for XML documents and XQuery item sequences.

The mapping from normalized XQuery Core to relational algebra by means of inference rules formalizes the compilation scheme and serves as basis for explaining the concepts of the transformation. From the inference rules we also derive the mapping from normalized XQuery Core to our target language MIL. Different optimizations increase the performance of the semantically correct but sometimes inefficient translation. Amongst others, an extension of the staircase join algorithm, which efficiently evaluates XPath location steps, enables us to exploit its techniques in the domain of XQuery. Another important optimization is the join recognition that, based on normalized XQuery Core patterns, detects relational joins and emits appropriate join plans.

Experiments not only justify the optimizations, but also demonstrate the outstanding scaling of our approach. An extensive performance comparison with other XQuery processors (using the XMark benchmark) furthermore marks the effectiveness of the approach. Finally, a conclusion sums up the ideas developed in this thesis and provides an outlook for the future topics in the course of the Pathfinder project.

Chapter 1

Introduction

Pathfinder is a research prototype whose task is to compile XQuery into a target language that can be interpreted by a relational database system. Its main focus lies on the development of ideas, which allow the efficient execution of the emitted query plans. Since the project was started at the University of Konstanz, Germany, in 2001, not only the XQuery language evolved, but also Pathfinder — researchers from the University of Twente, The Netherlands, and the Center for Mathematics and Computer Science (CWI) in Amsterdam, The Netherlands, joined the effort. Together many new ideas were developed (*e.g.*, *XPath Accelerator* [13], *staircase join* [17], and *XQuery on SQL Hosts* [15, 16]).

The work, on which this thesis is based, is a full implementation of these ideas, thus proving their significance in practice. The result is MonetDB/XQuery — a prototype system, which uses the MonetDB RDBMS [2] as its relational back-end to query large (multi-gigabyte) documents in interactive time. MonetDB/XQuery is available in open source [22].

1.1 Outline

The introduction proceeds with overviews of the XQuery language, MonetDB and its low level interpreter language, which is the target language of the approach explained in the following, as well as the Pathfinder compiler and its integration into the MonetDB RDBMS. The second chapter provides insight into the relational storage of both shredded and generated XML documents and explains how the basic data model in XQuery, namely *ordered sequences of items* are encoded. Chapter 3 presents the mapping from normalized XQuery Core to relational algebra. It closely follows the mapping described in [15, 16] and therefore repeats the ideas, before applying them on the relational algebra used in MonetDB. Early versions of Pathfinder/MonetDB struggled with large overhead as well as huge memory and time consumption. Order awareness, join recognition, and exploiting more properties in path steps are the optimizations, which allowed us to overcome these problems. They are explained in Chapter 4. Several experiments demonstrate the impact of these optimizations in Chapter 5. Furthermore, a complete system comparison against the latest versions of Galax [11] and X-Hive [26] using the XMark benchmark shows our unprecedented performance on larger documents. The last chapter concludes this thesis and gives an outlook of the ongoing work.

1.2 The XML Query Language: XQuery

As increasing amounts of information are stored, exchanged, and presented using XML, the ability to intelligently query XML data becomes increasingly important. XQuery satisfies

that need and allows flexible access to a broad spectrum of XML information sources. XQuery is derived from an XML query language called Quilt [6], which in turn borrowed features from several other languages, including XPath 1.0, XQL, XML-QL, SQL, and OQL.

The basic building block of XQuery is the *expression*, which may be constructed from keywords, symbols, and operands. In general, the operands of an expression are other expressions. XQuery is a functional language, which means that expressions can be nested with full generality. Path expressions and *FLWOR* expressions are two of the most important constructs in XQuery and they are therefore explained in more detail. The discussion of other concepts, like atomic types, comparisons, computations, casting, type checking, conditionals, variables, and XML construction are beyond the scope of this work and therefore omitted here. To follow this work it is, however, reasonable to be familiar with these constructs. To understand some decisions in the sequel it may be even necessary to know some basics about the XQuery semantics (*e.g.*, node identity, document order, or subtree copy). For a full reference, please refer to [5].

Path Expressions

Path expressions can be used to locate nodes within XML documents. A path expression consists of a series of one or more steps, separated by "`/`". A path step returns a sequence of nodes that are reachable from a given element (the context node) via a specified axis. Such a step has two parts: an axis, which defines the *direction of movement* for the step, and a node test, which selects nodes based on their kind, name, and/or type annotation. For example, the step `child::book` selects the `book` element children of the context node: `child` is the name of the axis, and `book` is the name of the element nodes to be selected on this axis. All together there are 12 axes in XQuery, which are listed with their semantics in Table 1.1. As a path step may contain several context nodes, it retrieves the resulting nodes for every context node, combines them, and returns them with duplicates removed in the order they appear in the document.

Axis	Result nodes
v/child	child nodes of v
v/descendant	all nodes in the subtree of v
v/descendant-or-self	v itself and its descendants
v/parent	parent of v
v/ancestor	recursive closure of parent axis
v/ancestor-or-self	v itself and its ancestors
v/following	nodes following v in document order
v/preceding	nodes preceding v in document order
v/following-sibling	followings with the same parent as v
v/preceding-sibling	precedings with the same parent as v
v/self	v
v/attribute	attribute nodes of v

Table 1.1: Overview of axis semantics originated in context node v.

FLWOR Expressions

A *FLWOR* expression is a feature of XQuery that supports iteration and binding of variables to intermediate results. It is often useful to compute joins between two or more documents and to restructure data. The name FLWOR, pronounced *flower*, is suggested by the keywords `for`, `let`, `where`, `order by`, and `return`. The `for` clause in a FLWOR expression consecutively binds each item in an input sequence to a variable, called the tuple stream.

The `let` allows binding variables to additional sequences of items. The optional `where` clause serves to filter the tuple stream, retaining some tuples and discarding others. The optional `order by` clause can be used to reorder the tuple stream. The `return` clause constructs the result of the FLWOR expression. It is evaluated once for every retained tuple in the tuple stream using the respective variable bindings. The result of the FLWOR expression is an ordered sequence containing the results of these evaluations.

Query Q_1 is such a FLWOR expression, whose `for` loop consists of 5 iterations:

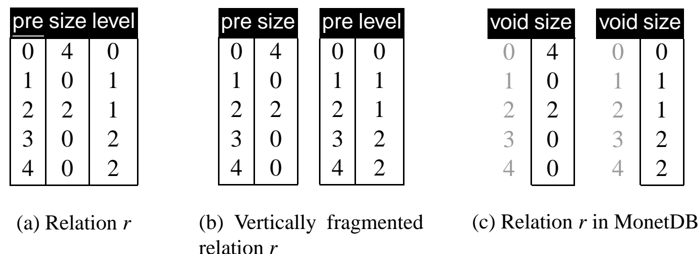
```
for $a in (8, 15, 12, 4, 9)
let $b := (string($a), "even")
where ($a mod 2 = 0)                .
order by $a ascending
return string-join($b, " is ")
```
(Q_1)

In every iteration `$a` gets bound to one value in the input sequence and a sequence of two strings (the string value of `$a` and `"even"`) is assigned to `$b`. The `where` clause filters all iterations where `$a` is bound to an even value and `order by` sorts the remaining values by their value. The `return` value is a concatenation of the strings in `$b` using the string `" is "` as delimiter. The result of the query is the item sequence: `("4 is even", "8 is even", "12 is even")`.

1.3 MonetDB

We chose to use the MonetDB RDBMS as our relational back-end for the Pathfinder compiler. MonetDB is an extensible main memory database system developed at the CWI in Amsterdam. Its aim to support multiple domains and to get the best performance out of modern CPU and memory hardware especially suits our demand. To reach these goals, MonetDB comes with some non-standard facilities.

One of these facilities is full vertical fragmentation, meaning that MonetDB only knows binary tables. The first column in such a table is called head, the second tail, and the complete table is named Binary Association Table (BAT). To avoid the loss of information, a fully vertically fragmented relation needs to save a unique key in each binary relation. This is shown in Figure 1.1(b), which introduces a new pre key column, after splitting relation r in Figure 1.1(a). The obvious performance penalty, which comes with the necessary key joins, is avoided in MonetDB by introducing virtual object identifiers (`void`) as keys. The columns of relations are saved in arrays allowing the `void` values to correspond to the offsets in the array (+ a given number). The storage of relation r in MonetDB consists of two arrays. The key column pre is not materialized anymore, because it matches the `void` column. To combine the columns in relation r, a positional lookup is used, which makes the key join almost a no cost operation.



pre	size	level
0	4	0
1	0	1
2	2	1
3	0	2
4	0	2

(a) Relation r

pre	size
0	4
1	0
2	2
3	0
4	0

pre	level
0	0
1	1
2	1
3	2
4	2

(b) Vertically fragmented relation r

void	size
0	4
1	0
2	2
3	0
4	0

void	size
0	0
1	1
2	1
3	2
4	2

(c) Relation r in MonetDB

Figure 1.1: Full vertical fragmentation in MonetDB.

MonetDB strictly follows a front-end/back-end architecture, which is introduced to reach the design goal of extensibility and to support multiple logical data models. A re-

lational front-end maps, *e.g.*, SQL queries into MonetDB requests — Pathfinder follows the same approach. The interface, to communicate with the underlying database kernel, is the low level interpreter language MIL[1]. Its table manipulation operations form a closed algebra on the binary model. Additionally, it is a computationally complete procedural language and allows extensions with new primitives, data types, and associated search accelerator structures. This extension mechanism will be used in Section 4.2 to add an efficient implementation of the staircase join algorithm in the programming language C. The remaining part of this chapter is used to shortly review some MIL operators.

MonetDB Interpreter Language (MIL)

The operators in MIL can be divided into two groups: table manipulation operations and programming language operators. The latter ones are inspired by the C programming language and contain, *e.g.*, variables, assignments, computations, comparisons, conditionals, loops, and functions. The former ones are basic database operations (*e.g.*, `join`, `project`, `select`, `unique`, `diff`, `union`, `intersect`, aggregates, ...) as well as additional operators, to support the binary data model. The basic database operators, which expect more than one BAT as input (like `join`), return again a binary relation, to fulfill the need for a closed algebra on the BAT type. In case of a join, this is achieved by projecting out the two join columns, retaining the other two columns.

`reverse`, `mirror`, and `mark` are specific operators implied by the binary table model. `reverse` switches head (the first column) and tail (the second column) of a BAT. This is necessary, because most operations (*e.g.*, `select`, `project`, aggregates) only work on the tail column. To combine multiple columns it is sometimes necessary to copy the key. This can be done using `mirror`, which copies the values of the head into the tail. `mark` is the operator, which creates a new key column. It consecutively assigns object identifiers (`oid`), which are numbers syntactically marked by "`@0`", starting at a given offset. This is also the mechanism to create `void` columns.

These three operators are for free as they modify only the descriptors of the BAT. For `reverse` the references to the arrays storing the values are switched and for `mirror` the references are copied. The `mark` operator only stores the offset in the BAT descriptor without materializing the enumerated column.

Figure 1.2 shows how relation *r* from Figure 1.1 can be generated: `bat(void, int)` creates a new BAT with the head type `void` and the tail type integer. `nil` stands for an undefined value; the dot concatenates multiple operations, and the semicolon finishes an expression. Because `mark` only works on tail values, a `reverse` is necessary twice, to create new numbers in the head column starting at offset 0.

```
> var r_size  := bat(void,int);
> var r_level := bat(void,int);
> r_size .insert(nil, 4).insert(nil, 0).insert(nil, 2).insert(nil, 0).insert(nil, 0);
> r_level.insert(nil, 0).insert(nil, 1).insert(nil, 1).insert(nil, 2).insert(nil, 2);
> r_size  := r_size .reverse().mark(0@0).reverse();
> r_level := r_level.reverse().mark(0@0).reverse();
```

Figure 1.2: Creation of relation *r* (see Figure 1.1) in MIL.

A special operator is the *multiplex* (`[f]`), which bridges the gap between the table manipulation and the procedural operators. `[f]` maps an operator `f` like *e.g.*, + to each row of a BAT. If such an operator expects more than one operand and input to the multiplex are multiple BATs, an implicit equi-join on the heads is performed.

Figure 1.3 continues the MIL example started in Figure 1.2. First we select the rows with size equal to `0` (discarding the first and the third row), copy the pre values into the tail and cast every row in the tail to an integer. The second row calculates the post values

[1] MIL stands for MonetDB Interpreter Language.

(pre/post encoding will be explained in the following chapter) using the multiplex operator. The implicit joins of [+] and [-] return only three rows, because of r_pre. The print performs an implicit join on the head values as well and therefore prints only three rows. The common head is printed as first column and the three remaining columns correspond to the tails.

```
> var r_pre := r_size.select(0).mirror().[int]();
> var r_post := r_pre.[+](r_size).[-](r_level);
> print (r_size, r_level, r_post);
#--------------------------------#
# t     tmp_214  tmp_190  tmp_241 # name
# void  int      int      int     # type
#--------------------------------#
[ 1@0,    0,       1,       0     ]
[ 3@0,    0,       2,       1     ]
[ 4@0,    0,       2,       2     ]
```

Figure 1.3: Calculate post value of all leaf nodes in relation *r*

A complete overview of MIL goes beyond the scope of this thesis. Most unexplained operators used in the next chapters, however, should be known either from relational algebra or from programming languages. For interested readers, [2, 21] will provide more details.

1.4 Pathfinder

The Pathfinder XQuery compiler can be used as new front-end to the MonetDB RDBMS (sketched in Figure 1.4). It consumes an XQuery expression, which is parsed, normalized, and translated into XQuery Core. The Core expression is then simplified, type checked and optimized. The last step of the compilation is the MIL code generation, which is also the focus of this thesis. Other approaches, which translate XQuery Core to relational algebra or SQL instead, are currently investigated in the course of the Pathfinder research project.

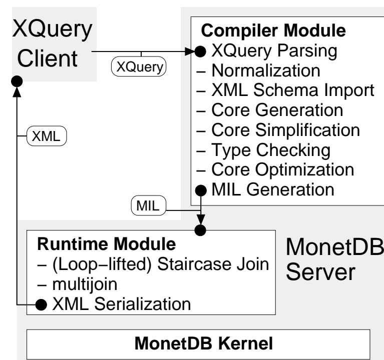


Figure 1.4: System architecture.

The MIL code generated by the Pathfinder compiler relies on some extensions added to the MonetDB back-end. The most prominent ones are the result serialization and the intelligent path step evaluation (staircase join). Others, however, will be mentioned in the course of this thesis.

Chapter 2

Relational Storage

A relational evaluation requires the tabular encoding of two principal data models. XML documents (or fragments thereof) form *ordered, unranked trees of nodes*. XQuery's processing model in turn is based on *ordered, finite sequences of items*. The first data model can be mapped using the pre/post encoding developed in [13]. It will be shortly summarized in Section 2.1, where we compare it with the pre/size encoding. The pre/size encoding is the mapping, which underlies our XML document storage and will be explained in detail in Section 2.2. The mapping of item sequences to relations as well as its actual storage is subject of Section 2.3 and Section 2.4, respectively.

2.1 pre/post Encoding

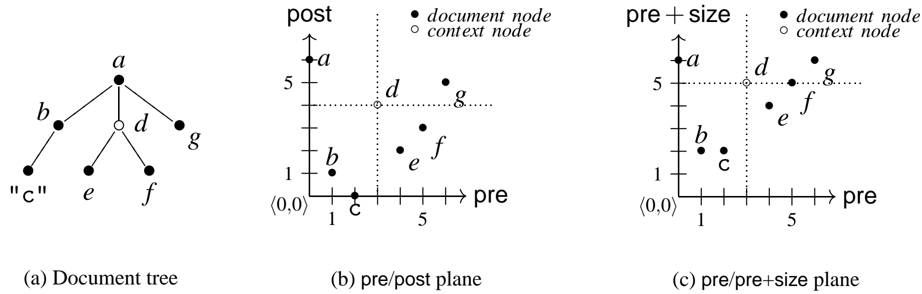
The pre/post encoding is a schema-independent mapping of XML documents into relations. It is a true isomorphism with respect to the tree structure. The encoding can be generated by counting the number of opening tags as pre values and the closing tags as post values during a sequential scan over the document. The pre and post values are then stored in a 2-column relation, where each row represents one node. The pre values alone already reflect the *document order* as well the *node identity*, which are two characteristics required in XQuery. Figure 2.1(a) shows the document tree of example document *Doc*:

```
<a><b>c</b><d><e/><f/></d><g a="42"/></a> .
```
(*Doc*)

Figure 2.1(b) shows the nodes plotted into a two-dimensional plane using the pre and post values to define their position. Node `d` for example resides at a pre value of 4 and a post value of 3. This tree encoding allows a highly efficient XPath processing. The staircase join [17] evaluates XPath location steps for a given sequence of context nodes in a single sequential scan over the tree encoding table. It works based on ranges on the pre/post relation visible in 2.1(b), where context node `d` divides the plane into four quadrants, which correspond to the four major XPath axes (`ancestor`, `descendant`, `following`, and `preceding`). Staircase join furthermore prunes the context node sequence to avoid overlap, partitions the document to completely avoid duplicate result nodes, and speeds up the processing by skipping considerable parts of the encoded tree, which do not contribute to the result.

The pre/post mapping records one more attribute (level), which stores the distance of a node to the root node. This additional column speeds up the processing of the `child`, `parent`, `following-sibling`, and `preceding-sibling` axes, which rely on the node depth of the context node.

In this work we use a different encoding variant by saving the pre values and the size of the subtree for each node. It is equivalent to the pre/post encoding, due to the fact that the post value can be recovered using the simple equation: post = pre + size - level.



(a) Document tree (b) pre/post plane (c) pre/pre+size plane

Figure 2.1: Graphical representations of document *Doc*

The pre/size mapping comes with similar characteristics like the pre/post mapping. Figure 2.1(c) shows *Doc* plotted on a pre/pre+size plan, where result nodes of the four major axes are still visible. But the pre/size encoding has even more advantages. To evaluate an XPath step, the pre/size encoding works with a small number of lookups (only lower boundary pre and upper boundary pre+size), whereas the pre/post encoding requires a post value comparison for each matching node to test the node containment. XPath location steps of the forward axes relying on the level information (`child` and `following-sibling`) in addition require only pre and size and therefore allow skipping. With the pre/post encoding, these axes require level, which prohibits skipping. To illustrate, for the `child` axis, we have that:

$$v \in c/\texttt{child} \Leftrightarrow$$
$$v_1.pre = c.pre + 1 \ \wedge \ v_1.pre \leqslant c.pre + c.size \ \cup$$
$$v_{i+1}.pre = v_i.pre + v_i.size + 1 \ \wedge \ v_{i+1}.pre \leqslant c.pre + c.size \ \ .$$

The last but perhaps most important advantage of the pre/size encoding is that element subtree copies, necessary for every element construction, are self containing in the pre/size encoding. The reason is that size is invariant with respect to copying and moving. In comparison, post values require updates for each subtree copy, because the new pre values automatically effect the post values.

2.2 XML Document Storage

The encoding described in the last section only saves the XML document structure. In this section the tree skeleton is enriched with the textual content of the document. Furthermore we explain the storage of multiple fragments and transient nodes created during querying.

Single Document Storage

The straightforward storage model would be one big relation holding all values of the five node kinds (*document*, *element*, *text*, *comment*, or *processing-instruction*):

pre	size	level	kind	prefix	uri	loc	text	comment	pi	target
2@0	0	2	text	null	null	null	c	null	null	null

Since every node has exactly one node kind, the records are heterogeneous. A fully decomposed storage model as described by Copeland and Khoshafian therefore seems to be a much better fit [8]. MonetDB internally already stores the relations columnwise and now additionally discards the large number of `null` tuples. In the following aligned columns like *e.g.*, prefix, uri, and loc are grouped to logical relations (*e.g.*, qn) whereas the underlying storage remains fully decomposed. The main table, in the following named pre|size

relation, holds the common values (pre, size, level, kind) and a foreign key ref, which together with kind refers to the values of the nodes. In addition, duplicate records in these aligned relations are eliminated to save storage space and to allow reference comparisons instead of string comparisons.

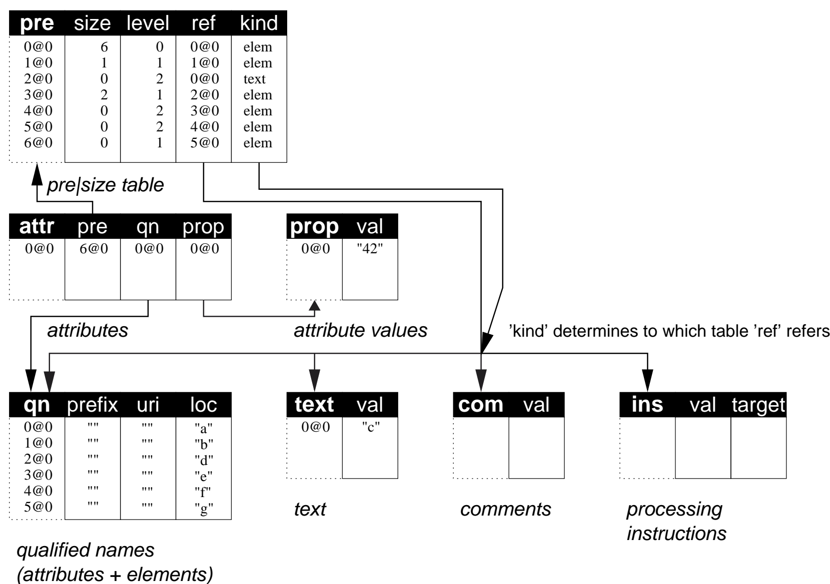


Figure 2.2: Relational storage of document *Doc* in MonetDB.

Figure 2.2 shows our relational storage, which stores document *Doc*. It contains two more relations (attr and prop), saving attributes and their content. The main reason behind this decision is the difference between attributes and other node kinds. Attributes do not follow a fixed order and have to be omitted during path steps. The column pre in relation attr saves the ownership of attributes as foreign keys. The names of attributes are saved in the qn relations together with the element names and referenced via the column qn. The relation prop saves the text content of the attributes (again kept unique).

Shallow Copying

One more reason, to use multiple relations instead of one big relation, is the copying effort necessary for a subtree copy of a node. With this storage scheme, we only have to copy slices of the shallow pre|size and attr relations. The copied rows contain only a small number of fixed size values and we avoid any string copying.

Multiple Documents and Transient Nodes

An XQuery expression may reference several documents in one query. Since we use a relational database, we do not want to shred a document for each query but use a persistent version of its shredded representation (see Figure 2.2). Multiple documents could be stored in one such set of document relations (in the following also called container), if a column frag was added to the pre|size relation to distinguish different documents. The drawback of this approach becomes clear immediately. For every XQuery expression all stored documents are loaded, even if only one small document is referenced. Each query furthermore has to copy or to lock the complete container during evaluation to add transient nodes. The same applies for storing additional XML documents.

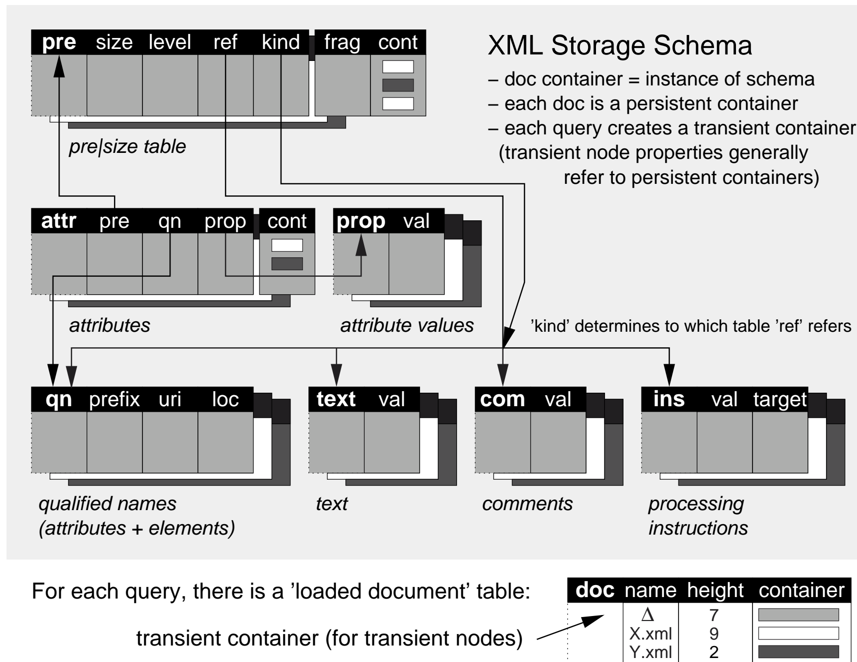


Figure 2.3: Horizontally partitioned XML storage in document containers.

The complete copying and loading overhead as well as the loss of concurrency can be avoided, if each shredded document resides on the disk separately according to the storage scheme sketched above. These document containers can be loaded separately during query evaluation and allow concurrent read operations as well. A transient document node container, which saves all nodes created during a query, also fits into this setup. The connection between these document containers builds a relation doc, which saves the document's name and exists for every query separately. With multiple referenced documents and each having pre values starting at 0, we loose the node identity. We overcome this problem by using the combination of doc and pre as our node identifier.

Figure 2.3 displays such an XML storage scheme for an example query with two loaded documents ($X.xml$ and $Y.xml$). For each XQuery expression an additional set of document tables (called Δ in Figure 2.3) is created, which saves the transient nodes constructed during query evaluation. In comparison to other containers, which encode always exactly one document, the transient document node container may contain multiple XML fragments. The additional frag column is used to mark the different fragments.

Multijoin

The shallow copying described in the last paragraph makes it necessary to add the cont column in the attr and the pre|size relation. It saves the document (or container) id, where the values of the nodes reside. This is the reason why the key of a node value lookup in the transient document node container now consists of ref, kind, and cont. To gather the node values of a set of nodes we added a new primitive *multijoin* to MonetDB, which does the lookup of the horizontally fragmented value tables efficiently. It groups the lookups by the document container and copes with the difference between the transient document node container and the other document containers. A more detailed description of *multijoin* will follow in Section 3.4.11.

The following XQuery expression Q_2 constructs new XML fragments, which generate

new nodes as well as a subtree copy of parts of *Doc*:

$$\texttt{(<h> doc("Doc")//d </h>, <i/>) .} \quad (Q_2)$$

Figure 2.4 illustrates the resulting pre|size table of the transient document node container. The names of the new nodes h and i are saved in the qn relation of the transient document node container. The entries for element d and its descendants in the pre|size relation of the document container of *Doc* are copied to the transient document node container and updated in the columns pre and level only. The other values (*e.g.*, the names) still reside in the document container of *Doc* (see column cont). The last row (element i) is in a different fragment and therefore has a different frag value.

pre	size	level	ref	kind	frag	cont
0@0	3	0	0@0	elem	0	Δ
1@0	2	1	2@0	elem	0	Doc
2@0	0	2	3@0	elem	0	Doc
3@0	0	2	4@0	elem	0	Doc
4@0	0	0	1@0	elem	1	Δ

Figure 2.4: pre|size table of the transient document node container of Q_2

2.3 Item Sequence Encoding

After fixing the encoding for the first datatype, this section now focuses on the second principle data type of XQuery: *ordered, finite sequences of zero or more items*. Assuming the underlying RDBMS provides polymorphic columns, a sequence of items can be easily saved in single column. The empty sequence then corresponds to an empty table and a single item maps to a relation with only one row. With such an encoding, we support finite sequences of zero or more items, but cannot rely on their order. [15, 16] suggest adding a second column pos, which stores the position of an item in the sequence. pos is a dense numbering starting at 1 for every sequence to support positional predicates (*e.g.*, `//d[2]`). The relation on the right encodes such a sequence `(2,"x",<a/>)`.

pos	item
1	2
2	"x"
3	<a/>

2.4 Item Storage

MonetDB does not offer polymorphic columns per se. We thus introduce a relation with the appropriate tail type for each type (and query). For attributes, other nodes, and QNames these relations are the document storage introduced in Section 2.2. An item is then implemented using a combination of reference and its type. Like in the document storage, a reference points to a value in its respective value relation. The entries of the value relations are kept unique to avoid multiple copies. *E.g.*, the query `for $a in 1 to 100 return "x"` saves "x" only once instead of 100 times. The relation on the right shows the corresponding string relation. The type (or value relation) a reference points to is encoded in a new column kind. Furthermore the attribute, node, and QName kinds are overloaded with the document id they refer to, therefore allowing to test node identity using item and kind.

ref	value
0@0	"x"

The example sequence from above (`(2,"x",<a/>)`) results in the relation depicted on the left. 2 is saved in the integer relation at offset 0@0, "x" in the string relation at offset 0@0, and the node <a/> is saved as first node in the transient document container Δ (again at offset 0@0).

pos	item	kind
1	0@0	integer
2	0@0	string
3	0@0	node(Δ)

Chapter 3

Core to MIL Translation

The relational storage structure explained in the previous chapter is suitable to support a fully relational XQuery evaluation engine. The following mapping, which relies on this storage scheme, is based on the compilation techniques described in [15, 16]. The starting point of the compilation is a normalized, simplified XQuery Core expression. Expressions, which are replaced during normalization are *e.g.*, the `where` clause and the comparisons operators with existential semantics (`=`, `!=`, `<`,...). Their replacements are additional `for` expressions, `if then else` constructs, functions like *e.g.*, `fn:empty`, and explicit comparisons (`eq`, `ne`, `lt`,...).

The mapping from normalized, simplified XQuery Core expression to a standard relational algebra is the subject of Section 3.4. The rules that describe the mapping process as well as the accompanying ideas were originally developed in [15, 16]. The *loop-lifting* concept, which builds the basis of the transformation, the relational operators, and the inference rule notation are recapped in Section 3.1, 3.2, and 3.3, respectively.

The inference rules in Section 3.4 are an extension of the rules in [15, 16] and serve as platform to illustrate the mapping process. Section 3.5 explains the application of the mapping using MIL as the relational target language.

3.1 Loop-Lifting

The iterative nature of the `for` expression in XQuery and a bulk-oriented relational processing appear to be contradictory. However as XQuery is side-effect free it is semantically sound to evaluate the loop body e for all iterations in parallel. This works by replacing all free occurrences of variable $\$v$ for all bindings in e by x_i ($e[x_i/\$v]$ denotes that substitution):

$$\texttt{for } \$v \texttt{ in } (x_1, x_2, \ldots, x_n) \texttt{ return } e \equiv$$
$$(e[x_1/\$v], e[x_2/\$v], \ldots, e[x_n/\$v])$$

With such a replacement, the iterations are avoided completely and a bulk-oriented processing is possible. In the following we describe a transformation, which implements the substitution. To distinguish separate logical iterations, we extend the pos|item [1] relation with a column iter, which stores the iteration number. Figure 3.1(a) shows the additional column iter attached to the encoded sequence expression `(2,"x",<a/>)`. The iter column contains the integer 1 in all three rows, stating that all items are in the same iteration (the initial iteration 1).

The variable binding of a `for` loop takes such a sequence as input and replaces the iter values with a new dense numbering, which respects the order given by the combination of

[1]Note: The item|kind representation is replaced by a polymorphic item column, again, for ease of readability. The mapping itself is not influenced by this change.

iter	pos	item
1	1	2
1	2	"x"
1	3	<a/>

(a) Input sequence (2,"x",<a/>)

iter	pos	item
1	1	2
2	1	"x"
3	1	<a/>

(b) Loop-lifted sequence ($a)

outer	inner
1	1
1	2
1	3

(c) Mapping relation

iter	pos	item
1	1	42
2	1	42
3	1	42

(d) Loop-lifted constant 42

iter	pos	item
1	1	42
1	2	42
1	3	42

(e) Backmapped body of for loop

Figure 3.1: Loop-lifting of $a and 42 in the query for $a in (2, "x", <a/>) return 42

old iter and pos column, starting at 1. The values in the pos column are then replaced by a 1, because every item is now a singleton sequence within its iteration. This mapping will be called *lifting* or *loop-lifting* in the following. It perfectly matches the semantics of the for expression: each item is bound in exactly one iteration. Figure 3.1(b) shows the binding of variable $a in the query for $a in (2,"x",<a/>) return 42. In the first iteration $a is bound to 2, in the second to "x" and to <a/> in the last.

All other variable bindings (*e.g.*, bound in a let clause or another for expression) and constants are loop-lifted using a map relation (outer|inner). This table records the just explained change of the iter values in the loop input sequence, where outer stores the iter before and inner the iter column after renumbering. A tuple $\langle o, i \rangle$ in this relation indicates that during the ith iteration of the inner loop body the outer loop body is in its oth iteration. The lifting proceeds in three steps: first an equi-join join between map and iter|pos|item, on the columns outer and iter, is performed. Then the outer and iter columns are removed and thirdly the column inner is renamed to iter. Figure 3.1(d) shows the lifted constant 42, which consisted of one tuple $\langle 1, 1, 42 \rangle$ before the join.

A similar mapping step is needed after the evaluation of the for loop body. The result has to be transformed into a sequence again. An equi-join of iter with the inner column of the map relation and the renaming of the outer column to iter performs the backmapping. Additionally a renumbering of the pos values is necessary. It takes the order of the old iter (inner) and pos columns into account and starts at 1 for each group defined by the new iter column (outer). Figure 3.1(e) shows the result of this backmapping step — the three item sequence (42, 42, 42), whose iter column is mapped back to 1 and the pos column to a dense enumeration.

Nested Scopes and Constant Expressions

The loop-lifting concept also works for nested for expressions. This is achieved by compiling each subexpression in dependence of all enclosing for loops. In example query Q_3 the loop body of the former example is replaced by a nested for loop:

$$s \begin{cases} \texttt{for \$a in (2, "x", <a/>) return} \\ \quad s_a \begin{cases} \texttt{for \$b in (10, 20) return} \\ \quad s_{a \cdot b} \left\{ \texttt{(\$a, 100)} \right. \end{cases} \end{cases} . \qquad (Q_3)$$

The map relation of the outer for loop stays the same (see Figure 3.1(c) and Figure 3.2(a)). It is used to lift the two constants (10 and 20), which form the input sequence of the inner for loop illustrated in Figure 3.2(b). This sequence is the basis for the new map relation of the second loop in Figure 3.2(c). To keep the two map relations apart we introduce a notion of scopes. For each loop body a new scope $s_{v_1 \cdot \ldots \cdot v_n \cdot v_{n+1}}$ is created, whose subscript is the subscript of the enclosing scope $v_1 \ldots v_n$ and the name of the for loop variable v_{n+1} (*e.g.*, s, s_a, $s_{a \cdot b}$ in Q_3). Now the map relations can be identified by attaching the scope information of the two scopes they connect. The map relation of the outer for expression in query Q_3 is then called map_{s,s_a} and the other $\mathsf{map}_{s_a,s_{a \cdot b}}$. Note that $\mathsf{map}_{s_a,s_{a \cdot b}}$ is the Cartesian product

outer	inner
1	1
1	2
1	3

(a) map_{s,s_a}

iter	pos	item
1	1	10
1	2	20
2	1	10
2	2	20
3	1	10
3	2	20

(b) `(10, 20)` in scope s_a

outer	inner
1	1
1	2
2	3
2	4
3	5
3	6

(c) $\mathsf{map}_{s_a,s_{a\cdot b}}$

iter	pos	item
1	1	2
2	1	2
3	1	"x"
4	1	"x"
5	1	<a/>
6	1	<a/>

(d) `$a` in scope $s_{a\cdot b}$

iter	pos	item
1	1	100
2	1	100
3	1	100
4	1	100
5	1	100
6	1	100

(e) `100` in $s_{a\cdot b}$

Figure 3.2: map relations and intermediate loop-lifted results of Q_3

of both loops, which is a consequence of this compilation scheme as well as of the XQuery semantics. Section 4.4 will describe how to avoid this cross product in certain cases.

Variable `$a` in query Q_3 is defined in scope s_a, but also visible in scope $s_{a\cdot b}$. To lift `$a` to scope $s_{a\cdot b}$ a join with $\mathsf{map}_{s_a,s_{a\cdot b}}$ is necessary — Figure 3.2(d) shows the result. For the second argument of the loop body, the constant `100`, the same mapping (join with both map relations) can be avoided. Because constants are always defined in the outer most scope s, it is valid to directly loop-lift them to the required scope by applying the Cartesian product of their pos|item tuple and a loop relation, which is the inner column of the respective map relation. In query Q3 the current loop relation $\mathsf{loop}_{s_{a\cdot b}}$ is the inner column of $\mathsf{map}_{s_a,s_{a\cdot b}}$ and the evaluation of the cross product with the constant `100` results in its loop-lifted representation displayed in Figure 3.2(e).

3.2 Relational Operators

In this section a short summary of the relational operators, which form the target language of the mapping in Section 3.4, is provided. Figure 3.3 lists the available operators, which

$\pi_{a_1:b_1,\ldots,a_n:b_n}$	projection (and renaming)
σ_a	selection
$\dot\cup$	disjoint union
$\times$	cartesian product
$\bowtie_{a=b}$	equi-join
$\setminus_{a_1,\ldots,a_n}$	difference
$max,\ sum_{b:\langle a\rangle/p}$	aggregates
a b	literal table
$\rho_{b:\langle a_1,\ldots,a_n\rangle/p,s}$	row numbering
$\circledast_{b:\langle a_1,\ldots,a_n\rangle}$	n-ary arithmetic/comparison operator $*$
$⏊_{\alpha,n}$	XPath axis join (axis α, node test n)

Figure 3.3: Operators of the relational algebra (a and b are column names).

are mostly variants of operators found in standard relational algebra [7, 10, 20]. While the projection operator π also supports renaming ($a_1 : b_1$ renames column b_1 to a_1), the join operator $\bowtie$ is restricted to equi-joins and the union operator $\dot\cup$ does not have to cope with duplicates. The difference operator returns all rows from the first argument, whose columns $a_1,\ldots,a_n$ have no matching tuples in the second argument. In addition to the normal aggregates (*e.g.*, *max*), which return a single value, we also support enhanced aggregates (*e.g.*, $sum_{b:\langle a\rangle/p}$) that partition a relation by a column p and evaluate within these partitions the aggregates on the column a. A binary relation, which stores for each partition the aggregated value (in column b), holds the result. The row numbering operator ρ is similar to the *DENSE_RANK* operator in SQL:1999. It is one of the most important operators in the

algebra, because XQuery heavily relies on order. $\rho_{b:\langle a_1,\ldots,a_n\rangle/p,s}(q)$ assigns each tuple in q a consecutive number, which is saved in column b. The constraint for the enumeration is the implicit order of q by the columns $a_1,\ldots,a_n$. Numbers start at offset s in each partition defined by the optional grouping column p. Operator $\circledast_{b:\langle a_1,\ldots,a_n\rangle}$ applies the n-ary operator $*$ to columns $a_1,\ldots,a_n$ and extends the input tuples with the result column b. The staircase join algorithm $⏋_{\alpha,n}$ supports the XPath axes. It calculates the result nodes for each iteration and returns them as an iter|item|kind relation, where all duplicates are removed.

3.3 Inference Rule Notation

To formally describe the mapping of XQuery Core to relational algebra we introduce a notation of inference rules. These inference rules are of the form:

$$\Gamma; \mathsf{loop}; \Delta \vdash e \Rightarrow (q, \Delta') \ .$$

The first argument of an inference rule denotes the variable environment Γ, which holds all visible variables and their algebraic expression. The second argument loop saves the current loop relation, to support efficient constant translation. The third argument stands for the container Δ, which is the transient document node container (see the relations in Figure 2.3) extended with the value relations (one for each kind). One specific characteristic of this container is that the value relations string and untypedAtomic as well as the document relations prop (attribute values), text (text content), and com (content of comments) are all represented by the same relation. The same holds for the value relation QName and the document relation qn. In both cases, this helps to minimize the copying effort necessary for node construction while still being able to reference the relations separately. Because the container Δ can be modified during evaluation (*e.g.*, by inserting values or by creating elements), it is also a return value of the inference rule (Δ'). The whole inference rule can be read as: Given Γ, loop, and Δ, the XQuery expression e compiles into the algebraic expression q with the (possibly) modified container Δ'.

Each XQuery compilation starts with an empty variable environment $\Gamma = \emptyset$, a singleton loop relation (loop = | iter | 1 |), empty relations in the container Δ, and the XQuery expression. All inference rules pass Γ, loop, and Δ top-down, while the resulting algebra expression is synthesized bottom-up. The result is a single algebra query that operates on the tree and sequence encodings sketched in Section 2.

All documents, which are referenced in the query are accessed read only and are therefore combined in a separate container doc[2]. doc and Δ both represent a set of tables. To reference one of their relations directly we introduce the notation *cont*[*name*], which references the *name* table of the container *cont*. *E.g.*, Δ[pre|size] references the pre|size table of the container Δ.

Additionally, inference rules use a notation to inspect static type information during compile time. Operator :: tests the static type of an expression during compile time. $e :: kind$ means that XQuery expression e has the static type *kind*. We denote,

$$\Delta' \equiv \Delta[\ldots, name \mapsto q, \ldots] \ ,$$

to make the modification of a relation in the container Δ and thus the modification of Δ explicit. It assigns the relation *name* of Δ the new relational representation q. The modified container Δ' records this side effect. All other relations of the container remain unaffected. Adding rows to an already filled relation can be done by union the representation before the replacement and the new tuples (*e.g.*, $q \equiv \Delta[name] \dot{\cup} tuples_{new}$).

[2]doc may contain multiple documents and thus represent multiple containers itself. This however is ignored, because the relations of doc are used only by special operators that are aware of this representation (staircase join and *multijoin*).

Thirdly the wrapper function *ref* is a shorthand for a much more space consuming algebra expression. *ref* modifies a container (first argument) by extending one of its relations with the values from the second argument. Then it extends the second argument with the references of the modified relation in the container. Depending on its subscript *ref* chooses a different implementation. The following equivalence rule illustrates the extended expression of *ref* with subscript integer:

$$\frac{\begin{array}{l} values_{new} \equiv unique\,(\Delta\,[\mathsf{integer}] \setminus_{value} q) \\ offset \equiv max\,(\Delta\,[\mathsf{integer}]) + 1 \\ tuples_{new} \equiv \pi_{ref,value}\left(\rho_{ref,offset}\,(values_{new})\right) \\ \Delta' \equiv \Delta\left[\ldots, \mathsf{integer} \mapsto \Delta\,[\mathsf{integer}]\,\dot{\cup}\,tuples_{new}, \ldots\right] \\ q' \equiv \Delta'\,[\mathsf{integer}] \bowtie_{value=value} q \end{array}}{(\Delta', q') \equiv ref_{\mathsf{integer}}\,(\Delta, q)} \;.$$

The two input arguments are the container Δ and a relation q that requires a column value. The result is a modified container Δ' and a relation q' that contains an additional column ref. Because the subscript of *ref* is integer, the integer relation of Δ gets extended by the values in the value column of q. This is done in the first four rows. The first rule determines the unique list of values that are not stored in the integer relation yet. The second rule determines the first free offset of the integer relation in Δ and the third enumerates the new values starting from this offset. Line four updates the integer relation as explained above to extend it with new values. The join in the last rule extends the second argument q with the references to the values in the modified container Δ'. With the wrapper function ref_{name}, we thus have a simple and elegant way to look up the references in the container relation *name* without paying attention to the necessary value insertion.

An example is the call of ref_{integer} with container Δ whose integer relation is empty. It results in a new tuple in the integer relation, whose reference is `0@0`, and the extended second input relation:

$$\left(\Delta', \begin{array}{|c|c|}\hline \text{ref} & \text{value} \\ \hline 0@0 & 2 \\ \hline \end{array}\right) \equiv ref_{\mathsf{integer}}\left(\Delta, \begin{array}{|c|}\hline \text{value} \\ \hline 2 \\ \hline \end{array}\right) \;.$$

3.4 Algebraic Translation Rules

Now all prerequisites are set up to translate XQuery to relational algebra. Every subsection in the sequel explains the transformation of one XQuery construct by means of an inference rule. Every rule compiles into a relational expression, which results in an iter|pos|item|kind relation[3]. While the inference rules exactly constitute the compilation scheme, the corresponding explanations will sometimes mix compile time and runtime. To motivate the relational expressions we describe their role during the evaluation. The mapping, however, compiles the complete query before evaluating the first operator. Note that the changes of the variable environment Γ only extend relational expressions. These expressions substitute the variables at compile time. The variable environment Γ therefore only helps to manage the relational expressions representing variables and is available at compile time only.

Query Q_4 combines constants, sequence construction, variable lookups, `let`, and `for` expressions. It will be the example, which helps to demonstrate the first mapping rules:

```
s  { let $a := (10, 20) return
   { for $b in (1, 2, 3) return .                    (Q4)
   {   s_b { ($a, $b)
```

[3]For certain scenarios another kind of interface may perform better. These alternatives will be the focus of Section 4.3.

Q_4 first binds the sequence `(10, 20)` to the variable `$a` and then iterates over the sequence `(1, 2, 3)` (bound to `$b`). It returns the concatenation of `$a` and `$b`, which is the sequence `(10, 20, 1, 10, 20, 2, 10, 20, 3)`.

3.4.1 Constants

The translation of constants is depicted in inference Rule CONST and starts with retrieval of the type information of constant c, followed by the generation of the relational expression that looks up the reference of c in the value relation *kind* (see extension of *ref* in the previous section). The second equivalence rule uses a cross product to attach the columns pos and kind to the returned reference. The compilation is completed by introducing a cross product between the relational expression in q_{res} and the loop relation. During the evaluation of such a relational expression the iter column lifts the constant c to the current scope. In addition to the compiled expression, the CONST rule also returns the (possibly) modified container Δ_1.

$$\frac{\begin{array}{l} c :: kind \\ (\Delta_1, q) \equiv ref_{kind}(\Delta, \begin{array}{|c|}\hline \text{value} \\ \hline c \\ \hline \end{array}) \\ q_{res} \equiv \begin{array}{|c|c|}\hline \text{pos} & \text{kind} \\ \hline 1 & kind \\ \hline \end{array} \times (\pi_{item:ref}\, q) \end{array}}{\Gamma; \text{loop}; \Delta \vdash c \mapsto (\text{loop} \times q_{res}, \Delta_1)} \quad \text{(CONST)}$$

If we pick the constant 3 from query Q_4, then the retrieved type is integer. The ref_{integer} function call adds 3 to the integer relation of container Δ and returns the reference $q \equiv \begin{array}{|c|c|}\hline \text{ref} & \text{value} \\ \hline 4@0 & 3 \\ \hline \end{array}$ (see integer table on the right). The extension of the reference q with pos and kind results in $\begin{array}{|c|c|c|}\hline \text{pos} & \text{item} & \text{kind} \\ \hline 1 & 4@0 & \text{integer} \\ \hline \end{array}$ and the cross product with loop ($\begin{array}{|c|}\hline \text{iter} \\ \hline 1 \\ \hline \end{array}$) produces $\begin{array}{|c|c|c|c|}\hline \text{iter} & \text{pos} & \text{item} & \text{kind} \\ \hline 1 & 1 & 4@0 & \text{integer} \\ \hline \end{array}$. Here the container Δ has been modified by the insertion of the tuple $\langle 4@0, 3 \rangle$ into the integer relation.

ref	value
0@0	10
1@0	20
2@0	1
3@0	2
4@0	3

3.4.2 Sequences

The SEQ rule concatenates exactly two sequences. Sequences with more than two items have to be compiled by applying Rule SEQ multiple times. The mapping first generates a relational representation of the first argument and then takes the modified container Δ_1 as input for the transformation of the second XQuery expression. The resulting algebra plans (q_1 and q_2) are merged with a disjoint union $\dot\cup$ after adding a column ord, which marks the difference between the two expressions. The ord column and the pos_1 column are the order constraints for the row numbering operator, which assigns new positions (pos) starting at 1 for each distinct iteration (iter). The role of the ord column is to ensure that every tuple of the first argument comes before the first tuple of the second argument within the same iteration. The pos column is used to maintain the original sequence order. The projection at the end removes the old pos and the ord column.

$$\frac{\Gamma; \text{loop}; \Delta \vdash e_1 \mapsto (q_1, \Delta_1) \qquad \Gamma; \text{loop}; \Delta_1 \vdash e_2 \mapsto (q_2, \Delta_2)}{\begin{array}{c}\Gamma; \text{loop}; \Delta \vdash (e_1, e_2) \mapsto \\ \left(\pi_{iter,pos:pos_1,item,kind}\left(\rho_{pos_1:\langle ord,pos\rangle/iter,1}\left(\left(\begin{array}{|c|}\hline \text{ord} \\ \hline 1 \\ \hline \end{array} \times q_1\right) \dot\cup \left(\begin{array}{|c|}\hline \text{ord} \\ \hline 2 \\ \hline \end{array} \times q_2\right)\right)\right), \Delta_2\right)\end{array}} \quad \text{(SEQ)}$$

The sequence construction of 10 and 20 is straightforward. The relational representations are prefixed with an ord column and concatenated with a union:

ord	iter	pos	item	kind
1	1	1	0@0	integer

$\dot\cup$

ord	iter	pos	item	kind
2	1	1	1@0	integer

$\equiv$

ord	iter	pos	item	kind
1	1	1	0@0	integer
2	1	1	1@0	integer

The row numbering operator creates a new position column containing the values 1 and 2 and the projection removes the ord column.

The more interesting sequence construction happens in the `for` loop body, where the loop-lifted representations of `$a` and `$b` are concatenated. The disjoint union just combines both extended item representation (see Figure 3.4). The row numbering starts for each iteration a consecutive sequence with the position 1. Because ord is the first order argument, all tuples from `$a` get the lower pos numbers (matching the order of the old pos column) and the tuples in `$b` then follow as third sequence item for all iterations (see the last column pos_1 in Figure 3.4).

`$a` :

ord	iter	pos	item	kind
1	1	1	0@0	integer
1	1	2	1@0	integer
1	2	1	0@0	integer
1	2	2	1@0	integer
1	3	1	0@0	integer
1	3	2	1@0	integer

$\dot{\cup}$

`$b` :

ord	iter	pos	item	kind
2	1	1	2@0	integer
2	2	1	3@0	integer
2	3	1	4@0	integer

$\equiv$

ord	iter	pos	item	kind	pos_1
1	1	1	0@0	integer	1
1	1	2	1@0	integer	2
1	2	1	0@0	integer	1
1	2	2	1@0	integer	2
1	3	1	0@0	integer	1
1	3	2	1@0	integer	2
2	1	1	2@0	integer	3
2	2	1	3@0	integer	3
2	3	1	4@0	integer	3

Figure 3.4: Sequence construction in loop body of query Q_4

3.4.3 Variable References

The sequence construction in Figure 3.4 just uses the relational representation of the variables `$a` and `$b`. They, however, are only available because the Rule VAR does produce the code. Rule VAR compiles a variable reference into a lookup of the variable $\$v$ in the environment Γ. This integrates the relational expression of $\$v$ stored in the variable environment Γ into the overall query plan. The correct loop-lifting of the variable $\$v$ is already encoded in its relational representation. A description, how variables are loop-lifted, will follow in Section 3.4.5.

$$\frac{}{\{\ldots,\$v \mapsto q_v,\ldots\};\text{loop};\Delta \vdash \$v \Mapsto (q_v,\Delta)} \quad (\text{VAR})$$

3.4.4 Let Bindings

The previous inference Rule VAR looks up variable representation in the variable environment Γ. The environment Γ contains these representations because every variable assignment in XQuery adds a new binding to Γ. The `let` expression is one XQuery construct, which introduces variable bindings.

Rule LET first compiles the `let` binding e_1 into its relational representation (q_1). For the compilation of the `let` body it extends the variable environment with the binding ($\$v \mapsto q_1$). During the compilation of e_2 the relational expression representing variable $\$v$ is therefore available. At runtime, there are no variable bindings anymore, because every reference was replaced by its relational expression (see also Rule VAR).

$$\frac{\Gamma;\text{loop};\Delta \vdash e_1 \Mapsto (q_1,\Delta_1) \qquad \Gamma + \{\$v \mapsto q_1\};\text{loop};\Delta_1 \vdash e_2 \Mapsto (q_2,\Delta_2)}{\Gamma;\text{loop};\Delta \vdash \texttt{let } \$v \texttt{ := } e_1 \texttt{ return } e_2 \Mapsto (q_2,\Delta_2)} \quad (\text{LET})$$

In query Q_4 the `let` expression compiles the two item sequence `(10, 20)` and adds its relational representation into the variable environment Γ. During the compilation of the

let body variable $a is looked up and its relational expression is included in the query plan (see Rule VAR).

3.4.5 For Expressions

The mapping of a for loop was already explained informally in Section 3.1. In this section the transformation is described again in more detail, adding also concepts, which were not discussed before. The inference Rule FOR maps the XQuery construct

$$\texttt{for}\ \$v\ \texttt{at}\ \$p\ \texttt{in}\ e_1\ \texttt{return}\ e_2$$

to its relational representation. The optional part 'at $\$p$' introduces variable $\$p$, which encodes the position of the current item (bound in $\$v$) in the input sequence (starting with the number 1).

The mapping starts with the compilation of the for loop binding e_1 that results in the relational expression q_1 (see line (1) in Rule FOR). The expression in the second line generates the loop numbering of the nested scope. The next equivalence rule uses this enumeration to encode the loop-lifted item representation of $\$v$ (one value per iteration).

The rules in line (4) and (5) translate the optional at $\$p$ part of the for expression. The position values of the compiled input expression e_1, q_1 match the required numbers. The pos values are therefore added to the integer relation of container Δ and their references together with the corresponding kind integer are assigned to q_p. Like the relational expression in q_v that represents the lifted variable $\$v$, q_p also encodes one tuple per iteration (pos $= 1$).

$$\frac{\begin{array}{ll}
(1) & \{\ldots,\$v_i \mapsto q_{v_i},\ldots\};\mathsf{loop};\Delta \vdash e_1 \Rightarrow (q_1,\Delta_1) \\
(2) & ext_v \equiv \rho_{inner:\langle iter,pos\rangle,1}(q_1) \\
(3) & q_v \equiv \begin{array}{|c|}\hline \mathsf{pos} \\ \hline 1 \\ \hline\end{array} \times \pi_{iter:inner,item,kind}(ext_v) \\
(4) & (\Delta_2, ref_p) \equiv ref_{\mathsf{integer}}(\Delta_1, \pi_{iter:inner,value:pos}(ext_v)) \\
(5) & q_p \equiv \begin{array}{|c|c|}\hline \mathsf{pos} & \mathsf{kind} \\ \hline 1 & \texttt{integer} \\ \hline\end{array} \times \pi_{iter,item:ref}(ref_p) \\
(6) & \mathsf{map} \equiv \pi_{outer:iter,inner}(ext_v) \\
(7) & \mathsf{loop}_v \equiv \pi_{iter:inner}(ext_v) \\
(8) & \{\ldots,\$v_i \mapsto \pi_{iter:inner,pos,item,kind}(q_{v_i} \bowtie_{iter=outer} \mathsf{map}),\ldots\} \\
 & +\{\$v \mapsto q_v\} + \{\$p \mapsto q_p\};\mathsf{loop}_v;\Delta_2 \vdash e_2 \Rightarrow (q_2,\Delta_3)
\end{array}}{\begin{array}{c}\{\ldots,\$v_i \mapsto q_{v_i},\ldots\};\mathsf{loop};\Delta \vdash \texttt{for}\ \$v\ \texttt{at}\ \$p\ \texttt{in}\ e_1\ \texttt{return}\ e_2 \Rightarrow \\ \left(\pi_{iter:outer,pos:pos_1,item,kind}\left(\rho_{pos_1:\langle iter,pos\rangle/outer,1}(q_2 \bowtie_{iter=inner} \mathsf{map})\right),\Delta_3\right)\end{array}} \quad \text{(FOR)}$$

The map relation in line (6) relates the iteration of the current scope (outer) with the iterations of the new scope (inner) and the expression in line (7) prepares the new current loop relation. The transformation of the loop body e_2 can be seen in line (8). All variable representations in variable environment Γ are lifted to the next scope by adding an equi-join with the map relation generated in line (6). The loop-lifting at this place ensures, that all visible variables are mapped correctly. Then the relational representations of the for loop variables $\$v$ and $\$p$ are added to the variable environment Γ to enable their reference lookups. loop_v becomes the new current loop relation and the (possibly) modified container Δ_1 completes the input for the compilation of the loop body e_2.

The relational representation of the loop body q_2 is mapped back to the enclosing scope by means of a join with the relation map (on iter and inner). The following row numbering ensures that each iteration in the outer scope has a consecutive pos value without changing the order of the inner scope (order by iter and pos).

Example query Q_4 starts the for loop evaluation with the translation of the input sequence e_1 resulting in q_1 depicted in Figure 3.5(a). The row numbering applied in line (2)

adds a new inner column, which matches the pos column of q_1 in Figure 3.5(a). q_v takes this inner column as new iter column and sets the positions (pos) to 1 (see Figure 3.5(b)). Because query Q_4 has no `at` clause, line (4) and (5) are omitted. The `map` relation is the combination of the iter columns of Figure 3.5(a) and Figure 3.5(b). The code generated for the loop-lifting in line (8) at runtime lifts the relational representation of variable `$a` from scope s to s_b (shown in Figure 3.5(c)). Similar to the `let` expression of query Q_4, the `for` expression extends the environment Γ with variable `$b` and its binding q_v at compile time. The relational representation is integrated into the query plan by the variable reference in the loop body. The evaluation of the `for` loop body processes the expression and returns the intermediate result (see Figure 3.4).

The backmapping join with `map` results in Figure 3.5(d) where all rows have an outer value of 1. The row numbering therefore interprets all rows as one partition and generates the new pos_1 values using the order of iter and pos. The projection builds the ubiquitous iter|pos|item|kind relation and completes the `for` loop evaluation.

iter	pos	item	kind
1	1	2@0	integer
1	2	3@0	integer
1	3	4@0	integer

(a) q_1

iter	pos	item	kind
1	1	2@0	integer
2	1	3@0	integer
3	1	4@0	integer

(b) q_v

iter	pos	item	kind
1	1	0@0	integer
1	2	1@0	integer
2	1	0@0	integer
2	2	1@0	integer
3	1	0@0	integer
3	2	1@0	integer

(c) `$a` in scope s_b

outer	iter	pos	item	kind	pos_1
1	1	1	0@0	integer	1
1	1	2	1@0	integer	2
1	2	1	0@0	integer	4
1	2	2	1@0	integer	5
1	3	1	0@0	integer	7
1	3	2	1@0	integer	8
1	1	3	2@0	integer	3
1	2	3	3@0	integer	6
1	3	3	4@0	integer	9

(d) Backmapping of loop body q_2

Figure 3.5: `for` loop mapping of query Q_4

3.4.6 If-Then-Else

The expression `if (`e_1`) then` e_2 `else` e_3 is the only conditional in normalized XQuery Core. Amongst others it *e.g.*, substitutes the `where` clause. Expression e_1 is evaluated first and returns a boolean value. If it holds true e_2 is evaluated otherwise e_3 is the result of the `if` expression. In this compilation scheme we cope with multiple `if` clauses (one for each iteration) at the same time.

Rule IF starts with the compilation of the boolean expression e_1. The result is split into two new loop relations (loop_2 and loop_3), which select all `true` and `false` values respectively. loop_2 is used as current loop relation for the compilation of e_2 and loop_3 as loop relation for the mapping of e_3. To ensure the correct representation of the variables in Γ a join with the corresponding loop relation is performed, which filters out all unnecessary iterations. The result is the union of both branches combining the iterations again.

$$\frac{\begin{array}{ll}
(1) & \{\ldots,\$v_i \mapsto q_{v_i},\ldots\};\text{loop};\Delta \vdash e_1 \Rightarrow (q_1,\Delta_1) \\
(2) & \text{loop}_2 \equiv \pi_{iter}\left(\sigma_{item=\texttt{TRUE}}\left(q_1\right)\right) \\
(3) & \begin{array}{l}\left\{\ldots,\$v_i \mapsto \pi_{iter,pos,item,kind}\left(q_{v_i} \bowtie_{iter=iter_1}\left(\pi_{iter_1:iter}\left(\text{loop}_2\right)\right)\right),\ldots\right\}; \\ \text{loop}_2;\Delta_1 \vdash e_2 \Rightarrow (q_2,\Delta_2)\end{array} \\
(4) & \text{loop}_3 \equiv \pi_{iter}\left(\sigma_{item=\texttt{FALSE}}\left(q_1\right)\right) \\
(5) & \begin{array}{l}\left\{\ldots,\$v_i \mapsto \pi_{iter,pos,item,kind}\left(q_{v_i} \bowtie_{iter=iter_1}\left(\pi_{iter_1:iter}\left(\text{loop}_3\right)\right)\right),\ldots\right\}; \\ \text{loop}_3;\Delta_2 \vdash e_3 \Rightarrow (q_3,\Delta_3)\end{array}
\end{array}}{\begin{array}{c}\{\ldots,\$v_i \mapsto q_{v_i},\ldots\};\text{loop};\Delta \vdash \texttt{if (}e_1\texttt{) then } e_2 \texttt{ else } e_3 \Rightarrow \\ \left(\left(q_2 \dot{\cup} q_3\right),\Delta_3\right)\end{array}} \quad (\text{IF})$$

Query Q_5 shows a modification of query Q_4, where an `if` expression is used. Instead of the concatenation of both variables `$a` is returned if its value is even and `$b` otherwise:

$$s \begin{cases} \texttt{let \$a := (10, 20) return} \\ \texttt{for \$b in (1, 2, 3) return} \\ \quad s_b \ \{\texttt{if (\$b mod 2 eq 0) then \$a else \$b} \end{cases} . \qquad (Q_5)$$

The result of the `if` clause is depicted in Figure 3.6(a), where the first and the third iteration return `FALSE`. loop_2 in line (2) of the IF rule contains only the tuple $\langle 2 \rangle$. It is used to discard all other iterations of the variables in line (3). Therefore the result in q_2 is the two item sequence shown in Figure 3.6(b). The evaluation of lines (4) and (5) returns the relational representation of iteration 1 and 3 accordingly (see Figure 3.6(c)).

iter	pos	item	kind
1	1	FALSE	boolean
2	1	TRUE	boolean
3	1	FALSE	boolean

(a) Mapping of `$a mod 2 eq 0`: q_1

iter	pos	item	kind
2	1	0@0	integer
2	2	1@0	integer

(b) Mapping of `$a` in `then` clause: q_2

iter	pos	item	kind
1	1	2@0	integer
3	1	4@0	integer

(c) Mapping of `$b` in `else` clause: q_3

Figure 3.6: `if` expression of query Q_5

3.4.7 Typeswitch

While most of the `typeswitch`es were removed during the simplification phase, some of them still remain. These are the ones, whose sequence type ty has to be decided dynamically at runtime. The type knowledge is wrapped in an operator *instanceof* representing some logic, which makes the runtime decision possible. It takes an item representation of the form iter|pos|item|kind and replaces the item values by the result of the comparison between type ty and the values in column kind. Having such a boolean mapping for the different iterations allows to apply the same compilation scheme as in Rule IF. The equivalence rules in lines (3)–(6) of Rule TYPESWITCH perfectly match the ones in lines (2)-(5) of the inference Rule IF in Section 3.4.6.

$$\frac{\begin{array}{ll}
(1) & \{\ldots, \$v_i \mapsto q_{v_i}, \ldots\}; \mathsf{loop}; \Delta \vdash e_1 \mapsto (q_1, \Delta_1) \\
(2) & inst_1 \equiv instanceof_{ty}(q_1) \\
(3) & \mathsf{loop}_2 \equiv \pi_{iter}(\sigma_{item=\texttt{TRUE}}(inst_1)) \\
(4) & \begin{array}{l}\{\ldots, \$v_i \mapsto \pi_{iter,pos,item,kind}(q_{v_i} \bowtie_{iter=iter_1} (\pi_{iter_1:iter}(\mathsf{loop}_2))), \ldots\}; \\ \mathsf{loop}_2; \Delta_1 \vdash e_2 \mapsto (q_2, \Delta_2)\end{array} \\
(5) & \mathsf{loop}_3 \equiv \pi_{iter}(\sigma_{item=\texttt{FALSE}}(inst_1)) \\
(6) & \begin{array}{l}\{\ldots, \$v_i \mapsto \pi_{iter,pos,item,kind}(q_{v_i} \bowtie_{iter=iter_1} (\pi_{iter_1:iter}(\mathsf{loop}_3))), \ldots\}; \\ \mathsf{loop}_3; \Delta_2 \vdash e_3 \mapsto (q_3, \Delta_3)\end{array}
\end{array}}{\begin{array}{c}\{\ldots, \$v_i \mapsto q_{v_i}, \ldots\}; \mathsf{loop}; \Delta \vdash \\ \texttt{typeswitch (}e_1\texttt{) case } ty \texttt{ return } e_2 \texttt{ default return } e_3 \mapsto \\ \left(\left(q_2 \dot\cup q_3\right), \Delta_3\right)\end{array}}$$

(TYPESWITCH)

3.4.8 Path Steps

The evaluation of path steps is encapsulated in the staircase join operator ⊿, which takes an iter|item|kind relation as input, evaluates the path step for all iterations, and returns again an iter|item|kind relation. Section 4.2 provides more details on the underlying algorithms.

Rule STEP starts with the compilation of the context set e. Its result is used as one input of the staircase join operator. $\sqsupset$ additionally uses the axis and node test information as input arguments as well as all available XML fragments. These fragments can be either documents (stored in doc) or transient nodes in the container Δ. Since the evaluation of an XPath step never escapes the fragment of its context node, it is possible to evaluate the step for each document in separation.

The result of the path step respects the XPath semantics and orders the resulting nodes according to their document order (depicted in the generation of column pos). The likewise required duplicate elimination is already applied inside the $\sqsupset$ operator.

$$\frac{\Gamma;\mathsf{loop};\Delta \vdash e \Mapsto (q_e,\Delta_1)}{\begin{array}{c}\Gamma;\mathsf{loop};\Delta \vdash e/\alpha::n \Mapsto \\ \left(\rho_{pos:\langle kind,item\rangle/iter,1}\left(\left(\pi_{iter,item,kind} q_e\right) \sqsupset_{\alpha,n} \left(\mathsf{doc} \dot\cup \Delta_1\right)\right),\Delta_1\right)\end{array}} \quad \text{(STEP)}$$

3.4.9 Text Constructor

In XQuery there exists the possibility to create new XML fragments with so called node constructors. These can be either direct constructors or computed ones. The first are XML nodes (*e.g.*, `<a/>`), while the latter ones use XQuery keywords. During normalization to XQuery Core all constructors are transformed into computed ones.

The next three sections will describe the mapping of `text`, `attribute`, and `element` constructors, respectively. The mapping of all other constructors is almost identical to the explained transformations and is therefore omitted.

The `text` constructor translates a string into a text node. In inference Rule TEXT the first step is the transformation of the strings (one in each iteration) to their relational representation q_1. Since the string relation and the text relation of container Δ both refer to the same table, q_1 automatically stores the correct references to the string representations of the text nodes. The equivalence rules in lines (2) and (3) emit the code, which determines the first free pre and the first free frag value. The following algebra expression in line (4) uses the results as starting points of the row numberings, which generate new pre and frag values for each text node. The frag values are incremented in parallel to the pre values, because the fragments of text nodes always consists of a single node (size = 0). The size information as well as the level, kind, and cont values are attached to the pre|ref|frag relation. Together they form the new entries for the pre|size relation of the container Δ representing the new text nodes ($nodes_{new}$ in line (5)). The insertion into the pre|size relation of Δ in line (6) makes them retrievable for later references. The result of Rule TEXT is the list of references (item) to the text nodes in the transient document node container Δ (kind) with one reference per iteration (pos = 1) as well as the modified container Δ_2.

$$\frac{\begin{array}{ll}
(1) & \Gamma;\mathsf{loop};\Delta \vdash e \Mapsto (q_1,\Delta_1) \\
(2) & offset_{pre} \equiv max\left(\pi_{pre}\left(\Delta_1\left[\mathsf{pre|size}\right]\right)\right)+1 \\
(3) & offset_{frag} \equiv max\left(\pi_{frag}\left(\Delta_1\left[\mathsf{pre|size}\right]\right)\right)+1 \\
(4) & q_3 \equiv \rho_{pre:\langle iter\rangle,offset_{pre}}\left(\rho_{frag:\langle iter\rangle,offset_{frag}}\left(q_2\right)\right) \\
(5) & nodes_{new} \equiv \begin{array}{|c|c|c|c|}\hline \mathsf{size} & \mathsf{level} & \mathsf{kind} & \mathsf{cont} \\ \hline 0 & 0 & \mathtt{text} & \Delta \\ \hline\end{array} \times \pi_{pre,ref:item,frag}\left(q_3\right) \\
(6) & \Delta_2 \equiv \Delta_1\left[\ldots,\mathsf{pre|size} \mapsto \Delta_1\left[\mathsf{pre|size}\right] \dot\cup nodes_{new},\ldots\right]
\end{array}}{\begin{array}{c}\Gamma;\mathsf{loop};\Delta \vdash \mathtt{text\{}e\mathtt{\}} \Mapsto \\ \left(\begin{array}{|c|c|}\hline \mathsf{pos} & \mathsf{kind} \\ \hline 1 & \mathtt{node(\Delta)} \\ \hline\end{array} \times \pi_{iter,item:pre}\left(q_3\right),\Delta_2\right)\end{array}} \quad \text{(TEXT)}$$

Query Q_6 shows an example, where an `text` constructor is applied. The variable `$a` is bound to the sequence `("one", "two", "three")` and for each item in this sequence a

text node with the respective textual representation is created:

```
for $a in ("one", "two", "three") return
    text {$a}
```
(Q_6)

The strings are stored in the string (text) relation using the references shown on the right. $offset_{pre}$ and $offset_{frag}$ both hold the value 1, since the pre|size relation in Δ_1 is empty and the new pre and frag values are 1,2,3 for both columns. The cross product with the columns size, level, kind, and cont is illustrated in Figure 3.7(a). The result of the text node construction, which holds the references to the new nodes can be seen in Figure 3.7(b).

ref	value
0@0	"one"
1@0	"two"
2@0	"three"

pre	size	level	ref	kind	cont	frag
1	0	0	0@0	text	Δ	1
2	0	0	1@0	text	Δ	2
3	0	0	2@0	text	Δ	3

(a) pre|size relation after insertion in Δ_3

iter	pos	item	kind
1	1	1	node(Δ)
2	1	2	node(Δ)
3	1	3	node(Δ)

(b) Resulting relation

Figure 3.7: Result of the `text` constructor in query Q_6

3.4.10 Attribute Constructor

The attribute construction has two input arguments: the attribute name and its value. After transforming both arguments (depicted in lines (1) and (2)), the relational representations implicitly — similar to the text construction — store references to the names in the qn relation and accordingly to the values in the prop relations of Δ. The determination of the first free attr value in the equivalence rule of line (3) is also similar to the text construction. The next step in the transformation is the combination of the name and value references in one relation (line (4)) followed by the generation of new attribute keys (attr) starting at $offset_{attr}$ (line (5)). The resulting relation first gets extended by a cont and a pre column to match the schema of the attr relation and is then added to the attr table of the container Δ. Since the attributes are not owned by elements, the pre column is empty. The cont column contains Δ, because the textual content (prop and qn) is stored in the transient document node container. The overall result of the `attribute` constructor is the iter|pos|item|kind relation holding the references to the new attributes and the extended container Δ_3 (saving the attributes).

$$\frac{\begin{array}{ll} (1) & \Gamma;\mathsf{map};\Delta \vdash e \mapsto (q_1,\Delta_1) \\ (2) & \Gamma;\mathsf{map};\Delta_1 \vdash e \mapsto (q_2,\Delta_2) \\ (3) & offset_{attr} \equiv max(\pi_{attr}(\Delta_2[\mathsf{attr}]))+1 \\ (4) & q_{qn\cdot prop} \equiv (\pi_{iter,prop:item}(q_{prop})) \bowtie_{iter=iter_1} (\pi_{iter_1:iter,qn:item}(q_{qn})) \\ (5) & q_{attr} \equiv \rho_{attr:\langle iter\rangle,offset_{attr}}(q_{qn\cdot prop}) \\ (6) & \Delta_3 \equiv \Delta_2\left[\ldots,\mathsf{attr}\mapsto \Delta_2[\mathsf{attr}] \dot\cup \left(\begin{array}{|c|c|}\hline pre & cont \\ \hline \texttt{null} & \Delta \\ \hline\end{array} \times \pi_{attr,prop,qn}(q_{attr})\right),\ldots\right] \end{array}}{\Gamma;\mathsf{map};\Delta \vdash \texttt{attribute}\ e_1\ \{e_2\} \mapsto \left(\begin{array}{|c|c|}\hline pos & kind \\ \hline 1 & \texttt{attr}(\Delta) \\ \hline\end{array} \times \pi_{iter,item:attr}(q_{attr}),\Delta_3\right)} \quad \text{(ATTR)}$$

Example query Q_7 is a modification of query Q_6 that replaces the `text` constructor with an `attribute` constructor. The name of the attributes is `number` for all three iterations:

$$s\begin{cases}\texttt{for \$a in ("one", "two", "three") return} \\ \quad s_a\ \texttt{\{attribute number \{\$a\}}\end{cases} \qquad (Q_7)$$

qn	prefix	uri	loc
0@0	""	""	"number"

(a) qn relation

prop	val
0@0	"one"
1@0	"two"
2@0	"three"

(b) prop relation

attr	pre	qn	prop	cont
1	null	0@0	0@0	Δ
2	null	0@0	1@0	Δ
3	null	0@0	2@0	Δ

(c) attr relation

Figure 3.8: Transient document node container Δ_5 after evaluation of query Q_7

The evaluation of the `attribute` constructor begins with the evaluation of the attribute names. Since the constant translation of `number` applies duplicate elimination, the name is added only once (see Figure 3.8(a)). The same is done with the values of the attributes using the prop relation as their storage (shown in Figure 3.8(b)). The join on iter combines the references to names and values. The row numbering starting from $offset_{attr}$, which is 1, creates the new attribute key. An extension with the columns pre and cont completes the new attribute entries. These are appended to relation attr (see Figure 3.8(c)) and complete the modification of the transient document node container Δ. The resulting relation holds the reference to the new attributes in the ubiquitous iter|pos|item|kind representation.

3.4.11 Element Constructor

Similar to the `attribute` constructor, the `element` constructor specifies the name of the new element in its first argument and the content in the second. Unlike the `text` and `attribute` constructor, the `element` constructor builds not only a single node, but has to cope with structural information as well. It has to combine zero or more node fragments and is enriched with zero or more attributes. The XQuery semantics furthermore expect these attributes and other nodes (together with their subtrees) to be a new copy of the existing ones (the input of the element content).

Since elements can contain elements and attributes can be either attributes assigned to the new nodes or in the subtree of a content node, some naming confusion may arise. Therefore we will call nodes listed in the element content ”context root nodes” in the following. Together with all their subtree nodes they will be named ”context nodes” and the attributes of this context nodes will be titled ”context attributes” accordingly. The elements, which are created by the element construction, will be named root nodes. Their attributes (listed in the element content) will be called root attributes.

With the document encoding explained in Section 2, the compilation of the `element` constructor can be grouped into 5 phases. The first phase retrieves all context nodes with a path step starting from the context root nodes and modifies the level information. The second phase builds the relational encoding of the root nodes. The third merges root and context nodes, creates new pre values keeping the correct order, and inserts the copies into the pre|size relation of the container Δ. The last two phases create copies of the root and context attributes, update their foreign key pre, and append them to the attr table of Δ.

The `element` constructor uses the function *multijoin* mentioned in Section 2.2. It has three arguments, where the first argument is a set of relations (*e.g.*, the pre|size relations of all documents) and the second is the name of the join column of the first argument (*e.g.*, pre). The last argument is the second join relation, which contains at least an item and a doc column. The task of these two columns is to identify the equal entries (item) in the correct relation doc (*e.g.*, item refers to pre values in the pre|size relation of document doc). It returns a relation whose schema contains the columns of the relations grouped in the first argument and the columns of the second join relation (like in a normal join).

The inference Rule ELEM starts with the compilation of the second argument to its relational representation q_2. q_2 is split into attributes (line (21)) and other nodes (line (2)). The latter ones are the context root nodes. These are extended in line (2) with a column

$$\text{(1)} \quad \Gamma;\mathsf{loop};\Delta \vdash e_2 \mapsto (q_2,\Delta_1)$$

ContextNodes

$$\text{(2)} \quad ctx_{root} \equiv \rho_{key:\langle iter,pos\rangle,1}\left(\sigma_{kind=\texttt{node}}\ q_2\right)$$

$$\text{(3)} \quad ctx_{nodes} \equiv \left(\pi_{iter:key,item,kind}\left(ctx_{root}\right)\right) ⊿_{\texttt{descendant-or-self},\texttt{node()}} \left(\Delta_1 \dot\cup \mathsf{doc}\right)$$

$$\text{(4)} \quad nodes \equiv \Delta_1\left[\mathsf{pre}|\mathsf{size}\right] \dot\cup \mathsf{doc}\left[\mathsf{pre}|\mathsf{size}\right]$$

$$\text{(5)} \quad ctx_{root\cdot extended} \equiv multijoin\left(nodes, \mathsf{pre}, \pi_{key,iter,item,doc:kind}\left(ctx_{root}\right)\right)$$

$$\text{(6)} \quad ctx_{root\cdot level} \equiv \pi_{key,iter_{in}:iter,lev_{root}:level}\left(ctx_{root\cdot extended}\right)$$

$$\text{(7)} \quad ctx_{nodes\cdot level} \equiv \pi_{iter:iter_{in},item,doc:kind,lev_{root},key}\left(ctx_{nodes} \bowtie_{iter=key} ctx_{root\cdot level}\right)$$

$$\text{(8)} \quad ctx \equiv \pi_{iter,pre,size,level:lev_{new},ref,kind,cont,doc,key}\left(\oplus_{lev_{new}:\langle lev_{diff},one\rangle}\left(\begin{array}{|c|}\hline \mathsf{one} \\ \hline 1 \\ \hline\end{array} \times \left(\ominus_{lev_{diff}:\langle level,lev_{root}\rangle}\left(multijoin\left(nodes, \mathsf{pre}, ctx_{nodes\cdot level}\right)\right)\right)\right)\right)$$

RootNodes

$$\text{(9)} \quad \Gamma;\mathsf{loop};\Delta_1 \vdash e \mapsto (q_1,\Delta_2)$$

$$\text{(10)} \quad ctx_{count} \equiv count_{size:\langle iter\rangle/iter}\left(ctx\right)$$

$$\text{(11)} \quad root_{size} \equiv \pi_{iter_1:iter,size}\left(\left(\begin{array}{|c|}\hline \mathsf{size} \\ \hline 0 \\ \hline\end{array} \times \left(\mathsf{loop} \setminus_{iter} ctx_{count}\right)\right) \dot\cup\ ctx_{count}\right)$$

$$\text{(12)} \quad root \equiv \begin{array}{|c|c|c|c|c|c|}\hline \mathsf{pre} & \mathsf{level} & \mathsf{kind} & \mathsf{cont} & \mathsf{doc} & \mathsf{key} \\ \hline \texttt{nil} & 0 & \texttt{elem} & \Delta & - & -1 \\ \hline\end{array} \times \pi_{iter,size,ref:item}\left(root_{size} \bowtie_{iter_1=iter} q_{qn}\right)$$

NodeInsertion

$$\text{(13)} \quad offset_{pre} \equiv max\left(\pi_{pre}\left(\Delta_2\left[\mathsf{pre}|\mathsf{size}\right]\right)\right) + 1$$

$$\text{(14)} \quad res \equiv \rho_{pre_{new}:\langle iter,ord,key,pre\rangle,offset_{pre}}\left(\left(\begin{array}{|c|}\hline \mathsf{ord} \\ \hline 1 \\ \hline\end{array} \times root\right) \dot\cup \left(\begin{array}{|c|}\hline \mathsf{ord} \\ \hline 2 \\ \hline\end{array} \times ctx\right)\right)$$

$$\text{(15)} \quad offset_{frag} \equiv max\left(\pi_{frag}\left(\Delta_2\left[\mathsf{pre}|\mathsf{size}\right]\right)\right) + 1$$

$$\text{(16)} \quad frag \equiv \pi_{iter_1:iter,frag}\left(\rho_{frag:\langle iter\rangle,offset_{frag}}\left(\mathsf{loop}\right)\right)$$

$$\text{(17)} \quad nodes_{new} \equiv \pi_{pre:pre_{new},size,level,ref,kind,frag,cont}\left(frag \bowtie_{iter_1=iter} res\right)$$

$$\text{(18)} \quad \Delta_3 \equiv \Delta_2\left[\ldots, \mathsf{pre}|\mathsf{size} \mapsto \Delta_2\left[\mathsf{pre}|\mathsf{size}\right] \dot\cup nodes_{new}, \ldots\right]$$

RootAttributes

$$\text{(19)} \quad attrs \equiv \Delta_3\left[\mathsf{attr}\right] \dot\cup \mathsf{doc}\left(\mathsf{attr}\right)$$

$$\text{(20)} \quad offset_{attr} \equiv max\left(\pi_{attr}\left(\Delta_3\left[\mathsf{attr}\right]\right)\right) + 1$$

$$\text{(21)} \quad attr_{root\cdot extended} \equiv multijoin\left(attrs, \mathsf{attr}, \pi_{iter,item,doc:kind}\left(\sigma_{kind=\texttt{attr}}\ q_2\right)\right)$$

$$\text{(22)} \quad attr_{root} \equiv \left(\pi_{iter_1:iter,pre_{new}}\left(\sigma_{level=0}\ res\right)\right) \bowtie_{iter_1=iter} attr_{root\cdot extended}$$

$$\text{(23)} \quad attr_{root\cdot new} \equiv \pi_{attr:attr_{new},pre:pre_{new},qn,prop,cont}\left(\rho_{attr_{new}:\langle attr\rangle,offset_{attr}}\left(attr_{root}\right)\right)$$

$$\text{(24)} \quad \Delta_4 \equiv \Delta_3\left[\ldots, \mathsf{attr} \mapsto \Delta_3\left[\mathsf{attr}\right] \dot\cup attr_{root\cdot new}, \ldots\right]$$

ContextAttributes

$$\text{(25)} \quad attr_{ctx\cdot extended} \equiv multijoin\left(attrs, \mathsf{pre}, \pi_{item:pre,pre_{new},doc}\left(\sigma_{level\neq 0}\ res\right)\right)$$

$$\text{(26)} \quad offset_{attr\cdot 2} \equiv offset_{attr} + count\left(attr_{root}\right)$$

$$\text{(27)} \quad attr_{ctx} \equiv \rho_{attr_{new}:\langle attr\rangle,offset_{attr\cdot 2}}\left(attr_{ctx\cdot extended}\right)$$

$$\text{(28)} \quad \Delta_5 \equiv \Delta_4\left[\ldots, \mathsf{attr} \mapsto \Delta_4\left[\mathsf{attr}\right] \dot\cup \left(\pi_{attr:attr_{new},pre:pre_{new},qn,prop,cont}\left(attr_{ctx}\right)\right), \ldots\right]$$

$$\Gamma;\mathsf{loop};\Delta \vdash \texttt{element}\ e_1\ \texttt{\{}e_2\texttt{\}} \mapsto \left(\begin{array}{|c|c|}\hline \mathsf{pos} & \mathsf{kind} \\ \hline 1 & \texttt{node}(\Delta) \\ \hline\end{array} \times \pi_{iter,item:pre_{new}}\left(\sigma_{level=0}\ res\right), \Delta_5\right)$$

(ELEM)

key that stores keys generated from the unique combination of iter and pos values. Using

this column key as new iter column ensures that no duplicates are removed within the ⍊ operator in line (3). The staircase join operator with the axis `descendant-or-self` returns all context nodes. The sole task of the next five equivalence rules (line (4)–(8)) is the update of the level column. Since the level of context nodes is arbitrary and the direct `child` of a new element is always in level 1, the level of all context nodes has to be changed in dependence of their context root node. That means the level of each context node has to be subtracted by the level of its context root node and added by 1. The algebra expression in line (5) looks up the level for all the context root nodes using a *multijoin* with the pre|size relations of all nodes. A join of the context root nodes with the context nodes on key maps the original iter value as well as the level of the context root nodes (lev_{root}) to all subtree nodes. The *multijoin* in line (8) enhances the context nodes with their node information and the $\ominus$ as well as the $\oplus$ operation update the level information.

The next phase of Rule ELEM creates the new element nodes specified in the `element` constructor (root nodes). The first task is the compilation of the tagnames and the second task is the retrieval of the size values, which are the number of subtree nodes. Counting the context nodes within each iteration provides the answer (see line (10)). This works because every iteration constructs exactly one element. Since all iterations, which have no context nodes, are not listed in *ctx* (remember that an empty sequence corresponds to an empty relation), the expression in line (11) is needed. It determines all empty iterations using the difference of the current loop relation and the counted iterations (ctx_{count}) and adds them with an size of zero to ctx_{count}. The join on iter in line (12) and the extension with the columns pre, level, kind, cont, doc, and key collects the remaining attributes and aligns the root node representations (*root*) with the relational representations of the context nodes (*ctx*).

One of the missing attributes is the new pre column, which has to contain new unique preorder ranks representing the document order of the nodes. The expression in line (13) looks up the first available pre value in the transient document node container Δ and the union in line (14) merges context and root nodes. The resulting relation is the input for a row numbering, which creates the new pre values starting at $offset_{pre}$. The order of the pre values is chosen in such a way, that all nodes in a fragment are in the same range (iter), all root nodes occur before their context nodes (ord), every context subtree is within one group (key) and the old pre order is retained in these subtree fragments. Note that the dummy values of the root nodes in pre and key do not matter for the generation of the new pre values, since the order is already defined using iter and ord. The last necessary column for the insertion of the new nodes into the pre|size relation of the transient document node container is the column frag, which has to be a new available fragment id for each new root node (iteration). The following equivalence rules (lines (15)–(17)) compute the offset, generate the new numbers, and map these numbers to their corresponding nodes. The algebra expression in line (17) furthermore adds a projection with a renaming that prepares the nodes for their insertion into the container Δ. The equivalence rule in line (18) then extends the current pre|size relation with the new nodes.

Without context and root attributes the inference Rule ELEM would conclude with the generation of the node references similar to the other constructors. The only difference for the compilation of the intermediate results is that only the references to the root nodes are returned (level $= 0$).

With attributes, two more phases have to be added to the compilation process. The first one generates code that adds copies of the root attributes to the attr relation of Δ. It starts with the computation of the first free attr value and then looks up the attribute information using a *multijoin* between the set of attr relations and the references of the root attributes. The foreign key relationship stored in column pre is updated by applying a join on iter between the resulting nodes (*res*) and the attributes. A row numbering updates the attr values and an insertion into the attr relation of the transient document node container Δ completes the mapping for the root attributes.

The relational expression generated in the last phase copies all the attributes owned

by context nodes. Therefore all the attributes of the context nodes (level $\neq 0$) are looked up using a *multijoin* on the column pre in all attr relations. Here the reason why the doc column was retained during the merge of the context and root nodes becomes obvious. Another observation is that the *multijoin* in line (25) is a real join in comparison to the previous invocations where it applies only a positional lookup. Exactly like the context attributes before a new attr values are introduced by the row numbering operator, the old pre values are replaced by the new ones , and the updated attributes are added to the attr relation of Δ.

```
let $ctx := doc("Doc")/a/* return
for $a at $pos in ("one", "two", "three") return          (Q8)
    element {$a} {$ctx[position() mod 2 + 1 = $pos ]}
```

Query Q_8 iterates over the same three-item sequence as query Q_6 and uses the strings of this sequence items as new element names (`$a`). The variable `$ctx` saves the three-item sequence containing the nodes `b`, `d`, and `g` from the document *Doc*. These nodes are the content of the new elements, where the rather complicated predicate `[position() mod 2 + 1 = $pos]` assigns iteration 1 node `d` (with the reference `3@0`), iteration 2 the nodes `b` and `g` (references `1@0` and `6@0`, respectively), and iteration 3 no content. The serialized result of query Q_8 is the following sequence:

```
<one><d><e/><f/></d></one>,
<two><b>c</b><g a="42"/></two>, .
<three></three>
```

iter	pos	item	kind
1	1	3@0	*node*(*Doc*)
2	1	1@0	*node*(*Doc*)
2	2	6@0	*node*(*Doc*)

The starting point of the element construction is the container Δ, whose only non-empty table is the string relation with the strings of the `for` loop input and the container doc that contains the document *Doc* (see Figure 2.2). The evaluation of line (1) results in the relation q_2 depicted on the right. It is the input for the row numbering, which adds a new key column. This key ensures that no duplicate nodes are removed during the `descendant-or-self` path step. Figure 3.9(a) shows the result of the path step in line (3) (ctx_{nodes}). The algebra expression in line (4) retrieves the node information of the context root nodes. The level for all context root nodes is 1. Together with the original iter column they are mapped to the context nodes, which is displayed in Figure 3.9(b). The *multijoin* followed by the addition and the subtraction in line (4) then provides the updated level values.

The evaluation of the element names in line (9) looks up the variable `$a`. The count in line (10) takes the resulting context nodes and counts them for each iteration. The result is a relation where iteration 1 and 2 both have a counted size of 3. Iteration 3 is not listed in ctx_{count} and therefore added using a difference and an union operation. The algebra expression in line (12) combines names and size information and extends the resulting relation with the missing values (shown in Figure 3.9(c)).

Because the pre|size relation of Δ is empty, $offset_{pre}$ and $offset_{frag}$ in the third phase both hold the value 1. Figure 3.9(d) shows *res*, which is the concatenation of context and root nodes enhanced with new pre values. The generated frag column is identical to the iter column and, after its mapping, completes the new entries for the pre|size relation (see Figure 3.9(e)).

For query Q_8 the fourth phase works on empty relations, since no root attributes are present. The subtree copy of the context attributes on the other hand has to copy the attribute of node `g`. The *multijoin* with the pre values in the attr relations and the context elements ($\sigma_{level \neq 0} res$) in line (25) provides the attribute `a="42"`:

$ctx_{attr\cdot extended} \equiv$

attr	pre	qn	prop	cont	item	pre_{new}	doc
0@0	6@0	0@0	0@0	*Doc*	6@0	8@0	node(*Doc*)

iter	item	kind
1	3@0	node(Doc)
1	4@0	node(Doc)
1	5@0	node(Doc)
2	1@0	node(Doc)
2	2@0	node(Doc)
3	6@0	node(Doc)

(a) ctx_{nodes}

iter	item	doc	lev_{root}	key
1	3@0	node(Doc)	1	1
1	4@0	node(Doc)	1	1
1	5@0	node(Doc)	1	1
2	1@0	node(Doc)	1	2
2	2@0	node(Doc)	1	2
2	6@0	node(Doc)	1	3

(b) $ctx_{nodes \cdot level}$

pre	size	level	ref	kind	cont	iter	key	doc
nil	3	0	0@0	elem	Δ	1	-1	–
nil	3	0	1@0	elem	Δ	2	-1	–
nil	0	0	2@0	elem	Δ	3	-1	–

(c) $root$

pre_{new}	ord	pre	size	level	ref	kind	cont	iter	key	doc
1	1	nil	3	0	0@0	elem	Δ	1	-1	–
5	1	nil	3	0	1@0	elem	Δ	2	-1	–
9	1	nil	0	0	2@0	elem	Δ	3	-1	–
2	2	3@0	2	1	2@0	elem	Doc	1	1	node(Doc)
3	2	4@0	0	2	3@0	elem	Doc	1	1	node(Doc)
4	2	5@0	0	2	4@0	elem	Doc	1	1	node(Doc)
6	2	1@0	1	1	1@0	elem	Doc	2	2	node(Doc)
7	2	2@0	0	2	0@0	text	Doc	2	2	node(Doc)
8	2	6@0	0	1	5@0	elem	Doc	2	3	node(Doc)

(d) res

pre	size	level	ref	kind	frag	cont
1@0	3	0	0@0	elem	1	Δ
5@0	3	0	1@0	elem	2	Δ
9@0	0	0	2@0	elem	3	Δ
2@0	2	1	2@0	elem	1	Doc
3@0	0	2	3@0	elem	1	Doc
4@0	0	2	4@0	elem	1	Doc
6@0	1	1	1@0	elem	2	Doc
7@0	0	2	0@0	text	2	Doc
8@0	0	1	5@0	elem	2	Doc

(e) $nodes_{new}$

Figure 3.9: Intermediate results of the element construction in query Q_8

The insertion into the attr relation of the transient document node container Δ with updated pre values concludes the element construction.

The result is the iter|pos|item|kind representation storing the references to the root nodes (1@0, 5@0, and 9@0 with the kind node(Δ)).

3.4.12 Count

The counting of sequences in Rule COUNT is similar to the size determination of root nodes (lines (10) and (11)) in Rule ELEM. After the compilation of the subexpression e into q a count partitioned by the iter values creates the relation q_{count}. Similar to the element construction, empty iterations are added using a difference and a union. The integers storing the count information are added to the integer relation of Δ and their reference form the overall result.

The compilation of the function `fn:count` can be easily modified to support the other aggregate functions (*e.g.*, min, max, and sum). These functions only require an additional join with a value relation to retrieve the values.

$$\frac{\begin{array}{ll} (1) & \Gamma; \mathsf{loop}; \Delta \vdash e \mapsto (q, \Delta_1) \\ (2) & q_{count} \equiv count_{value:\langle iter \rangle / iter}(q) \\ (3) & res \equiv \left(\begin{array}{|c|}\hline \mathsf{value} \\ \hline 0 \\ \hline \end{array} \times (\mathsf{loop} \setminus_{iter} q_{count}) \right) \dot{\cup}\ q_{count} \\ (4) & (\Delta_2, q_{res}) \equiv ref_{\mathsf{integer}}(\Delta_1, res) \end{array}}{\Gamma; \mathsf{loop}; \Delta \vdash \texttt{count(}e\texttt{)} \mapsto \left(\begin{array}{|c|c|}\hline \mathsf{pos} & \mathsf{kind} \\ \hline 1 & \texttt{integer} \\ \hline \end{array} \times \pi_{iter, item:ref}(q_{res}), \Delta_2 \right)} \quad \text{(COUNT)}$$

If we once more modify query Q_6 and count the looping variable `$a` this will of course result in the sequence `(1,1,1)` (see query Q_9):

```
for $a in ("one", "two", "three") return
    count ($a)
```

. (Q_9)

The most interesting observation in query Q_9 is that the expression in line (4) adds the tuple $\langle$0@0, 1$\rangle$ only once. Without duplicate elimination this would result in a much bigger integer relation.

3.4.13 Arithmetic and Comparison Operators

The compilation of comparison and arithmetic operators is very similar. Here the rules PLUS and LESS represent all other calculations and comparisons. Both rules first compile their subexpressions e_1 and e_2. The next step is a join with the value relations to gather the values. The test in line (1) therefore decides, using the static type of e_1, which *kind* relation has to be chosen. A join on the iter values combines both relational representations. Note that this join in line (6) perfectly matches the XQuery semantics: If either argument is the empty sequence the join finds no matching tuple and returns an empty sequence as well. The n-ary operator (here $\oplus$ and $\circledless$) generates the result for all iterations.

$$\frac{\begin{array}{ll} (1) & e_1 :: kind \\ (2) & \Gamma; \mathsf{loop}; \Delta \vdash e_1 \mapsto (q_1, \Delta_1) \\ (3) & q_{1\cdot extended} \equiv \pi_{iter_1:iter, value_1:value}\left(q_1 \bowtie_{item=ref} \Delta_1[kind]\right) \\ (4) & \Gamma; \mathsf{loop}; \Delta_1 \vdash e_2 \mapsto (q_2, \Delta_2) \\ (5) & q_{2\cdot extended} \equiv \pi_{iter_2:iter, value_2:value}\left(q_2 \bowtie_{item=ref} \Delta_2[kind]\right) \\ (6) & res \equiv \oplus_{value:\langle value_1, value_2 \rangle}(q_{1\cdot extended} \bowtie_{iter_1=iter_2} q_{2\cdot extended}) \\ (7) & (\Delta_3, q_{res}) \equiv ref_{kind}(\Delta_2, res) \end{array}}{\Gamma; \mathsf{loop}; \Delta \vdash e_1 \texttt{ + } e_2 \mapsto \left(\begin{array}{|c|c|}\hline \mathsf{pos} & \mathsf{kind} \\ \hline 1 & kind \\ \hline \end{array} \times \pi_{iter:iter_1, item:ref}(q_{res}), \Delta_3 \right)} \quad \text{(PLUS)}$$

$$
\frac{\begin{array}{ll}
(1) & e_1 :: kind \\
(2) & \Gamma;\mathsf{loop};\Delta \vdash e_1 \Rightarrow (q_1,\Delta_1) \\
(3) & q_{1\cdot extended} \equiv \pi_{iter_1:iter,value_1:value}\left(q_1 \bowtie_{item=ref} \Delta_1[kind]\right) \\
(4) & \Gamma;\mathsf{loop};\Delta_1 \vdash e_2 \Rightarrow (q_2,\Delta_2) \\
(5) & q_{2\cdot extended} \equiv \pi_{iter_2:iter,value_2:value}\left(q_2 \bowtie_{item=ref} \Delta_2[kind]\right) \\
(6) & res \equiv \oslash_{value:\langle value_1,value_2\rangle}\left(q_{1\cdot extended} \bowtie_{iter_1=iter_2} q_{2\cdot extended}\right)
\end{array}}{\Gamma;\mathsf{loop};\Delta \vdash e_1\ \mathtt{lt}\ e_2 \Rightarrow \left(\begin{array}{|c|c|}\hline \mathsf{pos} & \mathsf{kind} \\ \hline 1 & \mathtt{boolean} \\ \hline\end{array} \times \pi_{iter:iter_1,item:value}(res),\Delta_2\right)} \quad \text{(LESS)}
$$

The only difference between arithmetic and comparison operators is the result type. Comparison operators have a `boolean` type and encode their result directly in the item value. In comparison, the arithmetic operators have the same result type (*kind*) as their input and need to store the values in the value relation *kind* before returning the references.

```
for $a in (10, 20, 30) return
    $a + 30
```

(Q_{10})

```
for $a in (10, 20, 30) return
    $a < 30
```

(Q_{11})

The only difference between Query Q_{10} and Q_{11} is the operator in the `for` body. Both operations have the input *kind* integer and the join relation in line (6) is the same as well (see Figure 3.10(a)). The n-ary operators add and accordingly compare the values in the two value columns for each row. The result column value for Rule PLUS can be seen in Figure 3.10(b) and the one for Rule LESS in Figure 3.10(c).

$iter_1$	$value_1$	$iter_2$	$value_2$
1	10	1	30
2	20	2	30
3	30	3	30

(a) $q_{1\cdot extended} \bowtie_{iter_1=iter_2} q_{2\cdot extended}$

value
40
50
60

(b) $\oplus$

value
TRUE
TRUE
FALSE

(c) $\oslash$

Figure 3.10: Intermediate results of the queries Q_{10} and Q_{11}

3.4.14 Order by Expression

The `order by` clause can be understood as an optional extension of the `for` expression similar to the `at` clause (see XQuery pattern of Rule ORDER BY). Its expressions $e_1,\ldots,e_n$ store for each iteration a singleton sequence, whose values are used to change the order of the resulting sequence e_{return}. The order is determined first by the values of e_1 and then the secondary orderings in $e_2,\ldots,e_n$.

Because the XQuery pattern in Rule ORDER BY includes a `for` loop, large parts of the inference rule are identical to the Rule FOR (see lines (1)–(9)). The explanations of these inference rules are therefore omitted here. The compilation of the `order by` clause starts with the current map relation map_0. Each expression e_k $(1 \leq k \leq n)$ is compiled and integrated applying the equivalence rules in lines (12)–(15). Every expression $e_1,\ldots,e_n$ is compiled in dependence of the inner-most scope. Its input arguments are the mapped variable environment Γ' as well as the loop relation loop_v. The kind lookup in line (13) determines for each expression e_k the value relation whose values are looked up in line (14). For each expression e_k the relational expression in line (15) then extends the current

map relation map_{k-1} with the value column retrieved in the previous equivalence rule. In comparison to the FOR rule, the Rule ORDER BY applies the backmapping join of the `for` loop on the extended map relation map_n and the result of the `return` clause. The row numbering uses the additional columns of the map relation map_n to create the positions within an iteration (outer) according to the values of the `order by` expression $(c_1, \dots, c_n)$, thus implicitly realizing the new order.

Instead of the keyword `ascending` in Rule ORDER BY, `descending` can be used. Its compilation requires a reverse sorting or a reverse row numbering operator. Additionally the `order by` copes with sorting criterions, which contain values only for some iterations. The iterations storing the empty sequence are treated as if they either hold the smallest (keywords `empty least`) or the biggest (`empty greatest`) values. For these cases, the mapping proceeds similar to the compilation of missing values in the Rule COUNT. The difference operator retrieves the iterations with empty sequences and combines them with the existing iterations, using either the minimum or maximum value of their domain as sorting criterion.

(1) $\{\dots, \$v_i \mapsto q_{v_i}, \dots\}; \mathsf{loop}; \Delta \vdash e_{in} \Mapsto (q_{in}, \Delta')$

(2) $ext_v \equiv \rho_{inner:\langle iter,pos\rangle,1}(q_{in})$

(3) $q_v \equiv \begin{array}{|c|}\hline pos \\ \hline 1 \\ \hline\end{array} \times \pi_{iter:inner,item,kind}(ext_v)$

(4) $(\Delta'', ref_p) \equiv ref_{\mathsf{integer}}(\Delta', \pi_{iter:inner,value:pos}(ext_v))$

(5) $q_p \equiv \begin{array}{|c|c|}\hline pos & kind \\ \hline 1 & \mathtt{integer} \\ \hline\end{array} \times \pi_{iter,item:ref}(ref_p)$

(6) $\mathsf{map} \equiv \pi_{outer:iter,inner}(ext_v)$

(7) $\mathsf{loop}_v \equiv \pi_{iter:inner}(ext_v)$

(8) $\Gamma_v \equiv \{\dots, \$v_i \mapsto \pi_{iter:inner,pos,item,kind}(q_{v_i} \bowtie_{iter=outer} \mathsf{map}), \dots\} + \{\$v \mapsto q_v\} + \{\$p \mapsto q_p\}$

(9) $\Gamma_v; \mathsf{loop}_v; \Delta'' \vdash e_{return} \Mapsto (q_{return}, \Delta''')$

orderby

(10) $\mathsf{map}_0 \equiv \mathsf{map}$

(11) $\Delta_0 \equiv \Delta'''$

for each e_k in $e_1, \dots, e_n$:

(12) $\Gamma_v; \mathsf{loop}_v; \Delta_{k-1} \vdash e_k \Mapsto (q_k, \Delta_k)$

(13) $e_k :: kind_k$

(14) $res_k \equiv \pi_{iter,c_k:value}(q_k \bowtie_{item=ref} \Delta_k[kind_k])$

(15) $\mathsf{map}_k \equiv \pi_{inner,outer,c_1,\dots,c_k}(\mathsf{map}_{k-1} \bowtie_{inner=iter} res_k)$

$$\frac{\text{(1)–(15)}}{\begin{array}{c}\{\dots, \$v_i \mapsto q_{v_i}, \dots\}; \mathsf{loop}; \Delta \vdash \mathtt{for}\ \$v\ \mathtt{at}\ \$p\ \mathtt{in}\ e_{in}\ \mathtt{order\ by} \\ e_1, \dots, e_n\ \mathtt{ascending\ return}\ e_{return} \Mapsto (\pi_{iter:outer,pos:pos_1,item,kind} \\ (\rho_{pos_1:\langle c_1,\dots,c_n,pos\rangle/outer,1}(q_{return} \bowtie_{iter=inner} \mathsf{map}_n)), \Delta_n)\end{array}} \quad \text{(ORDER BY)}$$

Query Q_{12} contains an `order by` pattern in the loop body of the first `for` loop:

s { s_a { $s_{a \cdot b}$ } } (Q_{12})

```
for $a in (30, 20) return
  for $b in (2, 3, 1)
    let $c := $a + $b
    order by $c ascending
    return $c
```

The Rule ORDER BY compiles the inner `for` loop, before mapping the expressions of the `order by` clause. Because `$c` is the only expression in the `order by` clause, the equivalence rules of lines (12)–(15) are applied only once. Because e_1 and e_{return} are both compiled in dependence of the inner-most scope $s_{a \cdot b}$ and both contain the same XQuery

iter	pos	item	kind
1	1	5@0	integer
2	1	6@0	integer
3	1	7@0	integer
4	1	8@0	integer
5	1	9@0	integer
6	1	10@0	integer

(a) `$c` in scope $s_{a \cdot b}$ (q_{return} and q_1)

ref	value
5@0	32
6@0	33
7@0	31
8@0	22
9@0	23
10@0	21

(b) Part of the integer relation

outer	inner	c_1
1	1	32
1	2	33
1	3	31
2	4	22
2	5	23
2	6	21

(c) map_1

iter	pos	item	kind
1	1	7@0	integer
1	2	5@0	integer
1	3	6@0	integer
2	1	10@0	integer
2	2	8@0	integer
2	3	9@0	integer

(d) Result of Rule ORDER BY

Figure 3.11: Intermediate results of the query Q_{12}

expression, their relational representations (q_{return} and q_1) are identical at runtime as well (see Figure 3.11(a)). The join of the algebra expression in line (14) with the integer relation depicted in Figure 3.11(b) prepares the new order column c_1. In the equivalence rule of line (15) this column then extends the `map` relation (see Figure 3.11(c)). During the backmapping step this extended `map` relation is joined with the `return` clause. The following row numbering then implicitly sorts each resulting item sequence according to the column c_1 by assigning new position values. The outer column furthermore ensures that the order of the outer `for` loop is not modified. Figure 3.11(d) shows the result of the Rule ORDER BY, which encodes the two item sequences `(31, 32, 33)` and `(21, 22, 23)` in the first and accordingly second iteration.

3.4.15 Functions

There are two different kinds of functions in XQuery: *built-in* functions and *user-defined* functions (UDF). The built-in functions extend XQuery with new functionalities. For many of these functions, there exists a relational representation (see, *e.g.*, function `fn:count` in Section 3.4.12). Others rely on operators, which evaluate a simple operation in a loop-lifted manner, like the $\oplus$ operator. The few remaining ones, which do not fit into these two groups, require a special treatment. The mismatch occurs because of the XQuery semantics, which, *e.g.*, introduce dependencies between successive sequence items.

The user-defined functions can be compiled by replacing the function call with the function body before the algebra code gets generated. This replacement goes along with the replacement of the function variables and some additional casts. While this only works for non-recursive UDFs, the compilation of recursive user-defined functions is possible in MonetDB and will be topic of Section 3.5.12.

3.4.16 Casts

`cast` is the last keyword in XQuery, which is missing in this compilation scheme. Because casts rely on the static input type as well as the target type, a large variety of relational expressions can be generated. The basic idea is to use the cast operators of the underlying database back-end to evaluate a `cast` expression. Using a selection σ for each distinct kind value splits up the different types. For each kind the values are retrieved and then casted to the target type. A concatenation of the new references then completes the cast.

3.5 Translation to MIL

With the inference rules described in the previous section, the XQuery Core to MIL mapping can be adopted almost without further changes. The compilation creates for each XQuery construct a block of MIL code, resulting in one large MIL script, which then can

be evaluated sequentially. As relations in MonetDB are fully vertically fragmented, the mapping to MIL dedicates large parts of the MIL code to management purposes (*e.g.*, alignment of relations). Another feature of MonetDB is the materialization of intermediate results. This enables us to bind these intermediate results to MIL variables and reuse the bindings (similar to the names in the equivalence rules). We thus avoid the repeated evaluation of algebra expressions. For frequently used expressions, like *e.g.*, loop this significantly increases the performance. The four variables `iter`, `pos`, `item`, and `kind` are used as interface between the different blocks by assigning the result of every evaluated XQuery construct to them. The iter as well as the pos column use `oid` values, because in MIL the type `oid` is more carefully tuned than the `integer` type (*e.g.*, `void`, `mark`,...). The same holds for the loop and `map` relations, which are generated by `mark` operations.

With each new scope new loop and `map` relations are introduced. Because the old ones may be required at a later point (*e.g.*, to map back the results), new variable names are required. A distinct number for each scope solves this problem. These numbers (*e.g.*, num_1 and num_2) are generated at compile time and then used as suffixes for variables names. Intermediate results are stored in such suffixed variables as well to avoid loosing their information during the evaluation of a second nested expression (*e.g.*, during sequence construction).

In the following we describe the compilation of the single XQuery constructs to MIL. As the generated MIL programs describe their task in a very explicit, assembly style manner, the resulting MIL code gets pretty large. The compilation of the element construction, which requires a page in Rule ELEM, *e.g.*, maps to about 300 operations in MIL. Most of the operations are no cost operations, like *e.g.*, `reverse`, `mark`, or positional joins on `void` columns that are used only to align the binary relations. A detailed listing of this code would thus not give any new insight into the matter. We therefore restrict ourselves in the following to the explanation of the more interesting MIL fragments and refer the interested reader to the Pathfinder implementation.

3.5.1 Constants

Constants are compiled almost identical to Rule CONST in Section 3.4.1. The first two lines in Figure 3.12 correspond to the wrapper function *ref* of Rule CONST. The first assignment inserts the constant 10 into the integer value relation. For other kinds the compilation would choose a different relation. Setting the `seqbase` to `nil` before and to `0@0` afterwards ensures that the head column is not materialized during the insertion. The next line assigns the row containing the value 10 and its key to the variable `ref`. The used selection returns exactly one row, because the key property of the `int_relation` (set during initialization) implicitly removes duplicates.

In comparison to Rule CONST, we avoid building the literal table, but use the current loop relation as new iter column. The other three columns pos, item, and kind are generated using the `project` operator, which creates a copy of the head column (in our case a void column with the key) and inserts the respective second argument in all rows of the tail. Therefore we also looked up the `itemID` (see row three in Figure 3.12) instead of building the cross product.

```
1  int_relation := int_relation.seqbase(nil).insert(nil,10).seqbase(0@0);
2  var ref := int_relation.select(10);
3  var itemID := ref.reverse().fetch(0);
4  iter := loop;
5  pos := iter.project(1@0);
6  item := iter.project(itemID);
7  kind := iter.project(integer);
```

Figure 3.12: MIL code generated for the integer constant 10.

3.5.2 Sequences

The compilation of a sequence construction to MIL is shown in Figure 3.13. The mapping generates almost 50 commands to translate the operations in Rule SEQ to MIL. The main reasons are the management overhead as well as the explicit order necessary for the row numbering operator. The compilation starts with the mapping of the first argument e_1 (depicted in line 1 of Figure 3.13). During evaluation, its result will be stored in the four BATs assigned to `iter`, `pos`, `item`, and `kind`. Because the compilation of the second argument overwrites these bindings, the assignments in lines 2–5 save the intermediate result in a new unique variable set before the second argument e_2 is mapped to MIL (see row 6).

```
1 { .... } # code generated for first argument e1
2 var iter_num := iter;
3 var pos_num := pos;
4 var item_num := item;
5 var kind_num := kind;
6 { .... } # code generated for second argument e2
7 if (iter.count() = 0) {
8     iter := iter_num;
9     pos := pos_num;
10    item := item_num;
11    kind := kind_num;
12 } else if (iter_num.count() != 0) {
13    var seqb := int(max(iter_num.reverse())) + 1;
14    iter := iter.reverse().[int]().[+](seqb).[oid]().reverse();
15    pos := pos.reverse().[int]().[+](seqb).[oid]().reverse();
16    item := item.reverse().[int]().[+](seqb).[oid]().reverse();
17    kind := kind.reverse().[int]().[+](seqb).[oid]().reverse();
18    var iter_new := iter.append(iter_num);
19    var pos_new := pos.append(pos_num);
20    var item_new := item.append(item_num);
21    var kind_new := kind.append(kind_num);
22    var ord_new := iter.project(1).append(iter_num.project(2));
23    var order := ord_new.reverse().sort().reverse();
24    order := order.CTrefine(pos_new);
25    order := order.mark(0@0).reverse();
26    iter := order.join(iter_new);
27    pos := iter.sort().mark_grp(iter.tunique().project(1@0));
28    item := order.join(item_new);
29    kind := order.join(kind_new);
30 }
```

Figure 3.13: MIL code generated for the sequence operator.

The following 6 rows (lines 7–12) make use of the programming language concepts in MIL. If at runtime one intermediate result is empty (meaning all iterations return an empty sequence) the sequence construction is avoided completely. Otherwise the sequence construction proceeds with the concatenation of both intermediate results. The generated code in row 13 looks up the first unused key of the first argument and adds its value to all keys in the second argument (row 14–17), thus creating overall unique keys. Rows 18–21 combine both arguments and row 22 creates the new column ord.

The remaining rows generate the MIL code to support the row numbering operator ($\rho_{pos_1:\langle ord,pos\rangle/iter,1}$) of the SEQ rule. The MIL equivalent of the grouped row numbering operator ρ is the operator `mark_grp`. This operator however relies on the internal order of its input BAT. Hence explicit sorting of the iter BAT is needed before the `mark_grp` can be applied. Primary sorting in MonetDB is done using the `sort` operator, which works on the values in the head (see line 23). The `CTrefine` operator in MIL applies secondary orderings without violating the primary order. It uses the values of the second argument to re-sort the groups of tuples in the first argument whose tail values are equal (see line 24). After sorting we add a new `void` key (in line 25) and replace the old one (line 26, 28,

and 29). The new iter BAT (with its order ensured) is the first argument of the `mark_grp` command. The second argument is an unique list of iterations, which encodes in the tail the first values of the numbering within each group — `1@0`.

3.5.3 Let Bindings and Variable References

The straightforward way to implement the variable mapping would be to paste the generated MIL code for a variable binding wherever the variable is used. Since this would result in evaluating each binding multiple times, we decided to store the variable representations like intermediate results. To avoid a large number of small BATs (four for every variable), which have to be lifted for every `for` loop translation, we decided to interpret the variable environment Γ described in the inference rules in a literal way. We introduce a variable environment Γ, which stores the evaluated relational representations of the variables at runtime using an vid|iter|pos|item|kind relation (in MIL the five respective BATs). New variables are added by inserting their evaluated relational representation extended with a unique variable identifier vid (see also Figure 3.14).

```
1 { .... } # code generated for variable binding e1
2 var vid := iter.project(vid);
3 varenv_vid_num  := varenv_vid_num.seqbase(nil).insert(vid).seqbase(0@0);
4 varenv_iter_num := varenv_iter_num.seqbase(nil).insert(iter).seqbase(0@0);
5 varenv_pos_num  := varenv_pos_num.seqbase(nil).insert(pos).seqbase(0@0);
6 varenv_item_num := varenv_item_num.seqbase(nil).insert(item).seqbase(0@0);
7 varenv_kind_num := varenv_kind_num.seqbase(nil).insert(kind).seqbase(0@0);
8 { .... } # code generated for return expression e2
```

Figure 3.14: MIL code generated for the `let` expression.

The variable reference is solved by a simple selection on the vid column using a given variable identifier (see row 1 of Figure 3.15). The MIL code "`.mark(0@0).reverse()`" in line 2 creates a new virtual key and the join with the other variable environment BATs produces the aligned iter|pos|item|kind representation.

```
1 var vid := varenv_vid_num.select(vid);
2 vid := vid.mark(0@0).reverse();
3 iter := vid.join(varenv_iter_num);
4 pos  := vid.join(varenv_pos_num);
5 item := vid.join(varenv_item_num);
6 kind := vid.join(varenv_kind_num);
```

Figure 3.15: MIL code generated for the variable lookup.

A single join between the map relation and the current variable environment lifts *all* variables. Note that similar to the relations map or loop we create a new variable environment for each scope. These different representations of Γ can be used to additionally filter out the variables, which are not used in a nested scope, thus avoiding unnecessary loop-lifting. The filter relation, which stores the pairs of scope and variable identifier, can be generated at compile time. Figure 3.16 shows the loop-lifting from a scope num_1 to a scope num_2. Lines 3 and 4 create the filter, by looking up all vids in `var_mapping` using a given `for` loop identifier (for_{id}), and rows 5–9 apply the filter, storing its result in a new set of variable environment BATs. Lines 11–13 lift the key to the next scope and lines 14 to 18 create the expanded (lifted) representation using a join on the old keys.

3.5.4 For Expressions

`for` expressions are mapped analogously to the transformations in the previous three sections (3.5.1-3.5.3). Like the sequence construction it prunes parts of the unnecessary MIL code at runtime. Thus the lifting as well as the evaluation of the return value can be skipped

```
1 # filtering out the variables not used in a deeper nesting
2 # (var_mapping stores relation between variable and nesting)
3 var vid_map := var_mapping.select(for_id).mirror();
4 vid_map := varenv_vid_num1.join(vid_map).mark(0@0).reverse();
5 var varenv_vid_num2  := vid_map.join(varenv_vid_num1);
6 var varenv_iter_num2 := vid_map.join(varenv_iter_num1);
7 var varenv_pos_num2  := vid_map.join(varenv_pos_num1);
8 var varenv_item_num2 := vid_map.join(varenv_item_num1);
9 var varenv_kind_num2 := vid_map.join(varenv_kind_num1);
10 # loop-lifting of the filtered variables
11 var outer_inner := outer_num2.reverse().join(inner_num2);
12 var iter_map := varenv_iter_num2.join(outer_inner);
13 iter_map := iter_map.mark().reverse();
14 varenv_vid_num2  := iter_map.join(varenv_vid_num2);
15 varenv_iter_num2 := iter_map.join(varenv_iter_num2);
16 varenv_pos_num2  := iter_map.join(varenv_pos_num2);
17 varenv_item_num2 := iter_map.join(varenv_item_num2);
18 varenv_kind_num2 := iter_map.join(varenv_kind_num2);
```

Figure 3.16: MIL code generated for lifting the variables from scope num_1 to num_2.

completely, if the loop binding contains only empty sequences. Furthermore, we map Cartesian products using the MIL primitive `project` as well as the explicit sorting followed by the `mark` or `mark_grp` to evaluate the row numbering ρ. The lifting of variables and the insertion of the loop variables into the variable environment follows the description given in previous section.

```
1 var values := iter_values.reverse().mark(nil).reverse();
2 int_relation := int_relation.seqbase(nil).insert(values).seqbase(0@0);
3 var ref := iter_values.join(int_relation.reverse());
4 iter := loop;
5 pos := iter.project(1@0);
6 item := loop.join(ref);
7 kind := iter.project(integer);
```

Figure 3.17: MIL code generated for the wrapper function ref_{integer} in a `for` loop.

The only unmentioned concept used in the `for` loop mapping is the function application of the wrapper function ref_{integer}. In comparison to the function call in the constant mapping (see Figure 3.12), it adds a whole relation instead of a single value to the integer relation. This can be seen in line 1 of Figure 3.17, where `iter_values` corresponds to the second argument of ref_{integer}. In the `int_values` BAT the key property in the tail is set, which implicitly triggers the duplicate elimination during the insertion. A single join on the values obtains the references (see line 3).

3.5.5 If-Then-Else

For the mapping of an `if` expression we are almost done if we re-use parts of the MIL code produced for a `for` expression. The most interesting part here is the additional pruning, which can be applied using MIL conditionals. After evaluating the conditional expression e_1 we check whether we have no true or no false values. This allows us to prune either the complete `then` or the complete `else` branch evaluation. In Figure 3.18 these tests are evaluated in lines 6 and 7 and their result is used to skip lines 11–18 and 20–25, respectively. Furthermore the variable mapping (lines 11–12 and 21–22) as well as the evaluation of the union of both intermediate results in row 28 can be skipped if either branch is empty.

3.5.6 Typeswitch

The `typeswitch` expression is compiled into almost the same plan as the If-Then-Else. The only modification of Figure 3.18 is an additional *instanceof* test in line 2. Depending

```
1 { ... } # code generated for conditional expression e1
2
3 var falses := kind.select(FALSE);
4 var trues := kind.select(TRUE);
5
6 var skip_then := trues.count() = 0;
7 var skip_else := falses.count() = 0;
8
9 if (not(skip_then))
10 {
11     if (not(skip_else))
12     { ... } # code generated for filtering out false iterations in Γ
13
14     { ... } # code generated for then expression e2
15
16     if (not(skip_else))
17     { ... } # store intermediate result in new variable set (num)
18 }
19 if (not(skip_else))
20 {
21     if (not(skip_then))
22     { ... } # code generated for filtering out true iterations in Γ
23
24     { ... } # code generated for else expression e3
25 }
26
27 if (not(skip_then) and not(skip_else))
28 { ... } # union both intermediate result into overall result
```

Figure 3.18: MIL code generated for the If-Then-Else expression.

on the static input type *ty instanceof* compiles into different MIL code.

Figure 3.19 shows the code emitted for the *instanceof* call with the type integer. The grouped count (`{count}`) in line 1 returns all iterations annotated with the number of occurrences in iter and the following equivalence test replaces the numbers by boolean values, which are true for the value 1 only. Line 3 selects the matching iterations, line 4 prepares them to be a filter on the kind column and row 5 evaluates the condition on the kind integer for the iterations with exactly one item. Together with the missing iterations stored in `iter_false` they build the result of the conditional expression.

```
1 var iter_bool := iter.reverse().{count}(loop.reverse())[=](1);
2 var iter_false := iter_bool.select(false);
3 var iter_true  := iter_bool.select(true);
4 var iter_true_key := iter.join(iter_true.mirror()).reverse();
5 var iter_kind_bool := iter_true_key.join(kind).[=](integer);
6 var iter_item := iter_kind_bool.union(iter_false);
7 iter := iter_item.mark(0@0).reverse();
8 pos := iter.project(1@0);
9 item := iter_item.[oid]().reverse.mark(0@0).reverse;
10 kind := iter.project(boolean);
```

Figure 3.19: MIL code generated for the function call $instanceof_{\text{integer}}(e_1)$.

3.5.7 Path Steps and Constructors

The compilation of path steps and constructors to MIL strictly follows the mapping rules provided in Sections 3.4.8–3.4.11. Path steps, *multijoin*s, and *max* are mapped to their corresponding primitives in MIL and will not be explained in more detail. The few remaining operators like $\oplus$, $\ominus$, and $\setminus$ will be described in the following sections.

3.5.8 Count

For the mapping of the built-in function `fn:count` MIL offers special support. The rather complicated compilation using a grouped count followed by a difference and a union (see lines (2) and (3) of Rule COUNT) matches a single operator in MIL. Line 2 of Figure 3.20 shows this grouped count operator. `{count}` uses a copy of the second argument as result relation and records in the tail the number of corresponding tuples in the head of the first argument. With the iter column as first argument and the loop relation as second argument, we even handle empty iterations correctly. The remaining operations (lines 3-9) apply the ref_{integer} function and prepare the result for any following XQuery expression.

```
1 { .... } # code generated for the argument e
2 var iter_count := {count}(iter.reverse(), loop.reverse());
3 var values := iter_count.reverse().mark(nil).reverse();
4 int_relation := int_relation.seqbase(nil).insert(values).seqbase(0@0);
5 var ref := iter_count.join(int_relation.reverse());
6 iter := loop;
7 pos := iter.project(1@0);
8 item := loop.join(ref);
9 kind := iter.project(integer);
```

Figure 3.20: MIL code generated for the built-in function `fn:count`.

3.5.9 Arithmetic and Comparison Operators

The arithmetic and the comparison operators rely on MonetDB's multiplex operator (see Section 1.3), which evaluates single item operations on a complete relation. It perfectly supports the loop-lifting of MIL functions like arithmetic and comparison operators. Its implicit equi-join between relations furthermore simplifies the mapping process.

The less-than comparison operation in Figure 3.21 compares two integer values in multiple iterations. Line 2 and 3 store the necessary information of the evaluated first argument. Line 6 creates a BAT for the first argument, which holds the iter values in the head and the integer values in the tail. Row 7 does the same for the second argument. The multiplex operator (`[<]`) alone performs the join and the n-ary comparison operation of line (6) in Rule LESS.

```
1 { .... } # code generated for first argument e1
2 var iter_num := iter;
3 var item_num := item;
4 { .... } # code generated for second argument e2
5
6 var fst_arg := iter_num.reverse().join(item_num).join(int_relation);
7 var snd_arg := iter.reverse().join(item).join(int_relation);
8 var iter_item := [<](fst_arg, snd_arg).[oid]();
9 iter := iter_item.mark(0@0).reverse();
10 pos := iter.project(1@0);
11 item := iter_item.reverse().mark(0@0).reverse();
12 kind := iter.project(boolean);
```

Figure 3.21: MIL code generated for $\circledless$ comparison on integers.

3.5.10 Order by Expressions

The compilation of an `order by` expression follows the ideas described in Rule ORDER BY. The main difference between the algebra and the MIL mapping is the refinement of the order after each of the n steps using the `CTrefine` operator[4]. This allows to prune the

[4]The functionality of `CTrefine` was already explained in Section 3.5.2.

evaluation of the remaining argument list $(e_{k+1},\ldots,e_n)$ as soon as there is a clear — and thus not modifiable — ordering.

3.5.11 Built-In Functions

Many of the built-in functions as well as the basic casts perform typical programming language operations. These are lifted with the help of the multiplex operator. The aggregate functions (*e.g.*, min, max, and sum) use a grouped aggregate like the `{count}` operator for their compilation. Most of the other built-in functions (*e.g.*, `fn:name`, `fn:string`,...) can be mapped to MIL equally well. The few remaining ones, which are hard to translate using MIL, are solved using new primitives, written in C.

3.5.12 User-Defined Functions

Section 3.4.15 already suggests replacing the function call of user-defined functions by their body. However, for recursive user-defined functions, this would lead to an infinite recursion at compile time. We thus map user-defined functions into MonetDB functions (using the MIL primitive `proc`) that happily deal with recursive invocations. We compile the body of an user-defined function in XQuery like every other XQuery expression. The main difference is that the compiled expression is a stand alone MIL script in the `proc` body, which returns an iter|pos|item|kind relation. To evaluate the function body in dependence of the current scope the current loop and map relations are arguments of the function. Furthermore the relational representation of the XQuery function arguments and the global variables are collected in a new variable environment. This environment is passed to the function body as another argument of the `proc`. It makes the relational representations of all visible variables available in the function body.

Chapter 4

Optimizations

Following the compilation scheme described in Chapter 3 we built a first working version of our XQuery compiler. It enabled us to generate and evaluate MIL code for large parts of the *XQuery Use Cases* [1] as well as all XMark benchmark queries [25]. While the compilation provided us with a semantically correct translation, it did not avoid obvious performance bottlenecks. The moderate performance was confirmed by the execution times resulting from the evaluation of the XMark benchmark on medium sized documents (11 MB and 110 MB). The most important bottleneck became obvious in XMark Queries 8–12, whose evaluation took minutes on the 11 MB document and did not finish on the 110 MB file. These five queries compile nested `for` loops into Cartesian products (due to the necessary loop-lifting) before applying a comparison that discards most of the tuples (much like a relational join). But even queries without implicit joins failed to scale linearly. One reason was the large set of sort operations each query contained. The other obvious problem were the path steps, which scaled above linear.

In this chapter we discuss four different optimizations, which helps us to overcome this performance problems and enable the implementation to exploit the scalability of the relational backend MonetDB. The optimization in Section 4.1 provides a solution to the sorting overhead introduced by the explicit ordering necessary for the row numbering operator. It describes the idea of an order aware implementation, which keeps all results sorted by iter and pos. The high costs for XPath location steps on increasing document sizes were triggered by the iterative evaluation of the path algorithms. A path step and therefore also a scan over the document was necessary for each iteration. The loop-lifted staircase join discussed in Section 4.2 allows to evaluate a path step on all iterations within one scan over the document. The third optimization in Section 4.3 avoids the storage of intermediate results in their value relations, whenever their values are used by the next expression (*e.g.*, in the construction of nested elements or input values of a function call). In Section 4.4 our fourth optimization detects a large number of implicit relational joins in the XQuery Core expression and overcomes the emerging performance bottleneck with a more intelligent MIL translation.

4.1 Order Awareness

The mapping of XQuery into relational query plans, as described in Chapter 3, maintains order information in the relations that encode the input XML documents and in the (intermediate) results of XQuery expressions. Document order is reflected by the node surrogates (preorder ranks). Likewise, sequence order is maintained in the iter and pos columns of the intermediate relational results. The mapping would allow keeping track of order information without the need for explicit sorting in the physical relational query plans. However, it turns out that all conceivable implementations of the row numbering operator ρ impose

some ordering on their input relation as a precondition. This may thus lead to a large number of sort operators in the physical plans.

We decided to minimize the number of sort operations by keeping the intermediate results sorted on iter and pos. This is done by introducing new logical MIL operators for selections, joins, and distinct unions. Their underlying physical algorithms all preserve the order. All other operators in MonetDB already maintain order.

In case of a selection, we limit ourselves to the use of the `ScanSelect` implementation. This is no limitation since indices are only created for persistent relations. XML documents, which are stored in such relations, are only accessed over the path step algorithms. But these can make use of all select implementation and thus exploit the indices, as long as the result is sorted on iter and pos again. Joins are restricted to `NestedLoopJoin`, `SortMergeJoin`, and `HashJoin` implementations, which all maintain the order of their left argument. Some implementations (*e.g.*, `NestedLoopJoin`) even keep the order of the inner relation as minor ordering.

4.1.1 Merged Union

The distinct union is the only operation that requires a new implementation. The normal `AppendUnion` implementation just concatenates two relations, which are sorted on iter and pos, resulting in a relation, which is only sorted on pos given a grouping by iter. While this is enough for the next row numbering, an explicit sorting is required for the row numbering in the back-mapping phase of the `for` loop or the result serialization.

Figure 4.1(a) shows the both input relations (`$a` and `$b`) of the sequence construction in Query Q_{13}:

```
let $a := (100, 200) return
for $b in (10, 20) return   .
    ($a, $b)
```
(Q_{13})

Figure 4.1(b) displays the result of the `AppendUnion`. The following `mark_grp` in Figure 4.1(c) can be applied without sorting. For the backmapping, however, a sort is required because the physical order (100, 200, 100, 200, 10, 20) does not match the sequence order (100, 200, 10, 100, 200, 20) and therefore the `mark_grp` would return a wrong result.

The new `MergedUnion` algorithm we will now introduce uses one column as merging criterion. It scans both input relations and merges them into a new result relation. Tuples with smaller values in the merge column appear before tuples with larger values. If tuples with equal merge values appear both in the first and the second relation, the tuples from the first relation are *always* generated first in the result. For the sequence construction we use the iter column as merge criterion. Because of the precedence of the first argument, we can actually omit the introduction of the ord column altogether, whose only purpose is to enforce left-before-right union order.

Figure 4.1(d) shows the result of the `MergedUnion` operation. Here the iter columns are the merge columns. The smallest iteration in both relations is 1 and all tuples of the first argument `$a` in this iteration (see the first two rows in Figure 4.1(d)) appear before the tuples of `$b` (see third tuple $\langle 1,1,10 \rangle$). The same applies for the following iterations (here only iteration 2). After the `mark_grp` operation in Figure 4.1(e) we once again have a result, which is sorted on iter and pos.

The new `MergedUnion` implementation allows us to avoid more explicit sorting operations at the price of a sequential scan of both input relations. One scenario where the `MergedUnion` implementation clearly pays off is the union of root nodes and content nodes in the element construction. The `mark` operation, which creates new pre values (see line (16) in Rule ELEM), requires the result of the union to be sorted. This is the case if we apply `MergedUnion`.

$a

iter	pos	item
1	1	100
1	2	200
2	1	100
2	2	200

$b

iter	pos	item
1	1	10
2	1	20

(a) Input relations

iter	pos	item
1	1	100
1	2	200
2	1	100
2	2	200
1	1	10
2	1	20

(b) Result of `AppendUnion`

iter	pos	item
1	1	100
1	2	200
2	1	100
2	2	200
1	3	10
2	3	20

(c) Result of sequence construction with `AppendUnion`

iter	pos	item
1	1	100
1	2	200
1	1	10
2	1	100
2	2	200
2	1	20

(d) Result of `MergedUnion`

iter	pos	item
1	1	100
1	2	200
1	3	10
2	1	100
2	2	200
2	3	20

(e) Result of sequence construction with `MergedUnion`

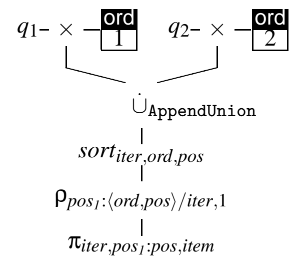
Figure 4.1: Sequence construction using different physical Union implementations

4.1.2 Conclusion

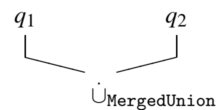
Replacing the logical operators allowed us to maintain the order on iter and pos for all iter|pos|item|kind representations. This leads to the minimization of the sort operations. A second and even more important result is that this physical order already encodes the pos information. We therefore can *completely* omit the pos column as well as the necessary row numbering operations. The few places where the positional information is required (*e.g.*, in the `at` clause of the `for` expression) a row numbering creates the pos column from scratch.

The order preservation additionally simplifies many mapping rules. *E.g.*, the sequence construction with order preservation consists of a single `MergedUnion`. The difference is shown by the relational plans on the right.

In some situations the positional information is not necessary at all. The XQuery keyword `unordered` is one example for such a case. The subexpressions of aggregations (*e.g.*, `fn:count`) also omit the pos column completely. In these situations our solution probably does not exploit the full strength of MonetDB. A mixed approach, which chooses the better physical representation at runtime, would be the best choice. However making the backend database aware of these order properties is beyond the scope of this thesis.



(a) sequence construction without order preservation

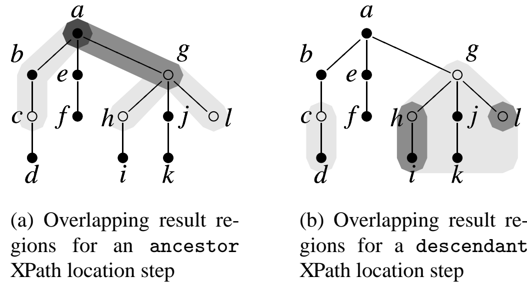


(b) sequence construction with order preservation

4.2 Loop-Lifted Path Steps

One way to evaluate path steps using the pre/size encoding, introduced in Section 2.1, are region scans. These scans can be evaluated in every database. The XPath semantics however require the result of a path step to be sorted in document order and duplicate free. Because fulfilling this additional requirements can become quite expensive, the staircase join was introduced in [17]. Staircase join evaluates a path step for multiple node preorder ranks (in the following called context nodes) in one sequential scan over the document. It exploits the tree knowledge of the XML documents and returns its result without duplicate nodes in document order. A more detailed description of staircase join will follow in Section 4.2.1.

For the evaluation of location path steps in XPath the staircase join is a good fit. It evaluates a path step starting from a set of context nodes within one scan over the docu-



(a) Overlapping result regions for an `ancestor` XPath location step

(b) Overlapping result regions for a `descendant` XPath location step

Figure 4.2: Intersection and inclusion of the `ancestor` and `descendant` paths of the context node sequence (c, g, h, l).

ment. But in XQuery path steps have to be evaluated for multiple context node sequences if they are nested inside `for` expressions. A loop-lifting of the staircase join therefore becomes necessary. Query Q_{14} illustrates a possible XQuery expression that embeds a `child` location step:

$$\begin{array}{l}\texttt{for \$}v\texttt{ in (}x_1\texttt{,}x_2\texttt{,}\ldots\texttt{,}x_n\texttt{) return}\\ \quad e(\texttt{\$}v)\texttt{/child::t}\end{array} \quad . \qquad (Q_{14})$$

The figure on the right shows the input relation of the `child` step, where the expression e evaluates each binding of `$`v into a sequence of node preorder ranks $(\gamma_{i,1}, \ldots, \gamma_{i,s_i})$ of length s_i $(1 \leqslant i \leqslant n)$. Because the basic staircase join cannot cope with iter values, alternatives have to be considered. The simplest idea would be to fall back to the regions scan, and eliminate the duplicates within iterations afterwards. Drawbacks are the size of the intermediate result, which may contain up to the number of document nodes multiplied by the number of iterations, and the costs of the duplicate elimination phase.

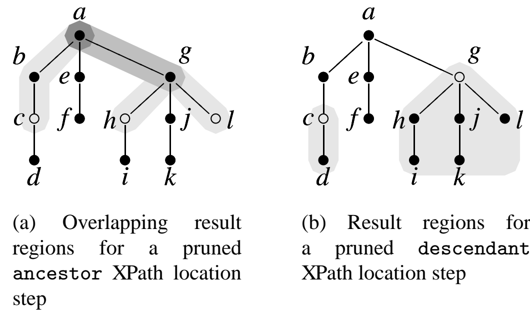
iter	pre
1	$\gamma_{1,1}$
1	$\gamma_{1,2}$
$\vdots$	$\vdots$
1	γ_{1,s_1}
$\vdots$	$\vdots$
n	$\gamma_{n,1}$
$\vdots$	$\vdots$
n	γ_{n,s_n}

A second alternative is the repeated application of the basic staircase join that evaluates an XPath location step for a single context node sequence (*i.e.*, for one of the above sequences $(\gamma_{i,1}, \ldots, \gamma_{i,s_i})$) during a single scan over a pre/size relation of an XML document. While this alternative is probably cheaper than the first approach, it still requires a sequential scan of the document for each iteration. For large documents and a large number of iterations this becomes expensive as well. The third alternative, which we will bring up in the sequel, is a new loop-lifted staircase join, which inherits many beneficial features of the basic staircase join and evaluates XPath location steps for multiple iterations in a single sequential scan over the document.

After describing the basic staircase join we will introduce loop-lifted variants of the staircase join for the `descendant` and `child` axes. Section 4.2.4 will describe how early kind and name tests are integrated in the staircase join and Section 4.2.5 will sketch the loop-lifted staircase join implementations for the remaining axes.

4.2.1 Basic Staircase Join

The staircase join allows to evaluate XPath location steps along arbitrary axes. It requires at most a single scan over document and context relation to produce a result that complies to the XPath semantics: duplicate free and sorted in document order. To achieve this, both input relations have to be sorted on the preorder ranks. In many cases this is no problem, since most path expression start at the root node and every following step returns its result nodes in document order. The staircase join relies on three techniques, namely *pruning*, *partitioning*, and *skipping*, which will be summarized in the following paragraphs. For more information please refer to [17].



(a) Overlapping result regions for a pruned `ancestor` XPath location step

(b) Result regions for a pruned `descendant` XPath location step

Figure 4.3: Pruning applied to the context node sequence (c, g, h, l) for the `ancestor` and the `descendant` paths.

Pruning Some nodes in the context node set of an XPath location step will never contribute to the result. These nodes lie in the result regions of some other context nodes. The result regions of an `ancestor` and a `descendant` step starting from the context node sequence (c, g, h, l) are illustrated by gray shadings in Figure 4.2(a) and Figure 4.2(b), respectively. The saturation depicts the overlap of the regions. In Figure 4.2(a) node g and its results is already included by the nodes h and l and in Figure 4.2(b) node g already spans the complete subtree including the context nodes h and l. By *pruning* the included context nodes from the context set, a large number of duplicate result values can be avoided. In case of the `ancestor` axis in Figure 4.2(a) node g can be pruned and for the `descendant` step in Figure 4.2(b) nodes h and l do not produce any new results. The result of pruning can be seen in Figure 4.3, where the more general effects of pruning are visible as well: For all axes except the `ancestor` and `parent` axes pruning removes *all* context nodes that introduce duplicates. The reason is the tree structure that prohibits partial overlaps for these axes. Only the `ancestor` and `parent` axes follow paths to the root and thus have common result nodes. Pruning therefore reduces only the number of context nodes but a partial overlap remains.

In the pre/size encoding, pruning can be evaluated efficiently by a simple comparison. In case of a `descendant` step, the pre, pre+size range of a context node ctx is used to prune another context node p: $ctx.\mathsf{pre} < p.\mathsf{pre} \leq ctx.\mathsf{pre} + ctx.\mathsf{size}$. To prune `ancestor` context nodes, we simply switch the condition and ancestor node p gets pruned: $p.\mathsf{pre} < ctx.\mathsf{pre} \leq p.\mathsf{pre} + p.\mathsf{size}$. The pre/size encoding for the XML tree in Figure 4.2 and 4.3 is listed on the right. The equation $6 < 7 \leq 6 + 5 = 11$ depicts the relationship between node g and node h. A similar condition also holds for node g and node l (pre value 11). A `descendant` step thus prunes node h and l and an `ancestor` step node g.

pre	size
0	11
1	2
2	1
3	0
4	1
5	0
6	5
7	1
8	0
9	1
10	0
11	0

Partitioning After pruning the context node sequence of the `ancestor` step, overlap and thus duplicate result nodes might remain (see Figure 4.3(a)). *Partitioning* the document along the preorder ranks of the context nodes into distinct intervals solves this problem (see Figure 4.4). Within each interval only one single context node is *active* and determines the result nodes. Each partition is scanned once in document order and the properties of the currently *active* context node are used to collect matching tuples. The single sequential scan ensures that no document node is returned more than once, thus avoiding all duplicates. It also guarantees that the result is generated in document order since both input relations of the staircase join are sorted in document order. All requirements of the XPath semantics are therefore satisfied and no post-processing is necessary.

Figure 4.4 illustrates the partitions introduced by the remaining context nodes of the `ancestor` step (c, h, and l). Node a is now generated by context node c only and appears only once in the result. The same applies for node g, which is generated by node h. Node l

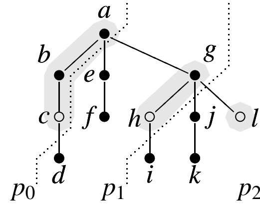
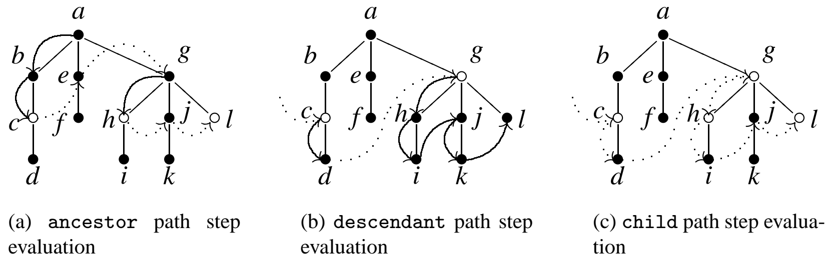


Figure 4.4: Partitioning applied to the context node sequence (c, h, l) for the `ancestor` paths.



(a) `ancestor` path step evaluation

(b) `descendant` path step evaluation

(c) `child` path step evaluation

Figure 4.5: Evaluation of XPath location steps with skipping starting from the context node sequence (c, g, h, l). (Skipping is depicted by the dotted arrows and scanning by the normal arrows.)

produces no new result within its partition p_2.

Skipping While pruning and partitioning already fulfill the requirements posed by the XPath semantics, more tree knowledge can be utilized to speed up the path step evaluation. One observation is that parts of the XML tree do not contain any result nodes. Scanning these possibly large parts is surely wasteful. Staircase join therefore stops the sequential scan and *skips* to the next interesting node[1]. In case of the `descendant` axis, skipping is applied directly after the last node in the subtree of the current context node. The jump target is the next context node. For the `ancestor` axis it is the next possible ancestor node in the document, thus skipping any preceding nodes. For a `child` path step, we know that the first child of a context node c (if it is not a leaf) has the pre value $v_1.\mathsf{pre} = c.\mathsf{pre} + 1$ and the next children are siblings of this child ($v_{i+1}.\mathsf{pre} = v_i.\mathsf{pre} + v_i.\mathsf{size} + 1$). As long as we do not reach the end of the subtree region ($c.\mathsf{pre} + c.\mathsf{size} + 1$) we thus can skip either to the next sibling omitting the complete subtree or to the next context node, which is then a node in the subtree.

Figure 4.5 demonstrates the skipping for the `ancestor`, `descendant`, and `child` path steps, whose context node sequence is the pruned node sequence (c, g, h, l). The `ancestor` location step starts scanning (↷) from the beginning of the document until it reaches node c. After node c, node h becomes active. The subtree of node c is skipped ($c.\mathsf{pre} + c.\mathsf{size} + 1$), because it contains only preceding nodes. The subtree of the next node (e) also contains preceding nodes only and is skipped as well. Now the scanning condition is fulfilled again and it proceeds with nodes g and h, where again skipping starts. The `descendant` path step in Figure 4.5(b) directly jumps to the first context node c, evaluates all its descendants ($c.\mathsf{pre} + c.\mathsf{size}$), and jumps to the next context node (g) where again all descendants are scanned. The evaluation of the `child` axis proceeds similarly. It also directly skips to node c and keeps skipping to the next children (here only node d). As soon as all children v_i are collected ($v_i.\mathsf{pre} > c.\mathsf{pre} + c.\mathsf{size}$) it jumps to the next context node. In Figure 4.5(c) the evaluation of node g is paused as long as the nested context node h collects its children and then proceeded.

[1]This is perfectly supported in MonetDB, where positional jumps correspond to an array lookup.

Basic Staircase Join Algorithms The staircase join implementations combine the three techniques pruning, partitioning, and skipping into a single sequential scan over the pre-sorted relations. Here, we shortly repeat the implementations for the `descendant`, `ancestor`, and `child` axes to allow a later comparison with their loop-lifted variants.

```
scj_desc (doc : TABLE(pre, size), ctx : TABLE(pre))
 BEGIN
   ASSERT (doc.pre IS DENSE AND ASCENDING);   /* for positional lookup */
   ASSERT (ctx IS SORTED ON (pre));           /* document order */
   result ← NEW TABLE(pre);
   nxtCtx ← 0;
   lstCtx ← SIZE(ctx);
   cur_node ← 0;
   WHILE (nxtCtx ≤ lstCtx) DO                 /* iterate over all context nodes */
     IF (cur_node ≤ ctx[nxtCtx].pre) THEN     /* pruning */
       cur_node ← ctx[nxtCtx].pre;            /* skipping */
       lst_node ← cur_node + doc[cur_node].size;
       cur_node ← cur_node + 1;               /* omit self node */
       WHILE (cur_node ≤ lst_node) DO         /* collect all descendant nodes */
         APPEND ⟨cur_node⟩ TO result;
         cur_node ← cur_node + 1;
     nxtCtx ← nxtCtx + 1;
   RETURN result;
 END
```

Figure 4.6: Staircase join: `descendant` axis.

Figure 4.6 shows the implementation of the `descendant` location step. Inside the outer loop, which iterates over the context node sequence, a conditional prunes the context nodes that are in the subtree of the outer context node. The pruned context nodes are discarded because the previous active context node already consumed the matching part of the document. Skipping is applied by jumping directly to the next context node and the inner loop partitions the document by iterating over all the descendant nodes v of the currently active context node c in the document ($c.\mathsf{pre} < v.\mathsf{pre} <= c.\mathsf{pre} + c.\mathsf{size}$).

```
scj_anc (doc : TABLE(pre, size), ctx : TABLE(pre))
 BEGIN
   ASSERT (doc.pre IS DENSE AND ASCENDING);   /* for positional lookup */
   ASSERT (ctx IS SORTED ON (pre));           /* document order */
   result ← NEW TABLE(pre);
   nxtCtx ← 0;
   lstCtx ← SIZE(ctx);
   cur_node ← 0;
   WHILE (nxtCtx ≤ lstCtx) DO                 /* partitioning */
     WHILE (cur_node < ctx[nxtCtx].pre) DO    /* iterate over document nodes */
       lst_node ← cur_node + doc[cur_node].size;
       IF (ctx[nxtCtx].pre ≤ lst_node) THEN   /* check ancestor property */
         APPEND ⟨cur_node⟩ TO result;
         cur_node ← cur_node + 1;
       ELSE
         cur_node ← lst_node + 1;             /* skip preceding nodes */
     nxtCtx ← nxtCtx + 1;
   RETURN result;
 END
```

Figure 4.7: Staircase join: `ancestor` axis.

The implementation of the `ancestor` axis in Figure 4.7 also consists of two nested loops. Similar to the `descendant` step, the nesting does not introduce multiple scans

but a merge-like sweep over the relations. In comparison to the `descendant` axis, the implementation of an `ancestor` step looks up the size information of the document nodes instead of the sizes of the context nodes. Another difference is the missing pruning in the implementation of the `ancestor` location step. To prune context nodes for this axis a reverse scan would be required, which is more expensive than splitting up the document into a larger number of partitions. The evaluation of the `ancestor` step in Figure 4.5(a) however still matches the algorithm (node *g* is active during skipping from node *d* to *g*).

```
scj_child (doc : TABLE(pre,size), ctx : TABLE(pre))
 BEGIN */
   ASSERT (doc.pre IS DENSE AND ASCENDING);       /* for positional lookup */
   ASSERT (ctx IS SORTED ON (pre));               /* document order */
   result ← NEW TABLE(pre);
   nxtCtx ← 0;
   lstCtx ← SIZE(ctx);
   active ← NEW STACK(eos,nxtChld);               /* stack of active context nodes */
   WHILE (nxtCtx ≤ lstCtx) DO                     /* iterate over all context nodes */
     IF (active IS EMPTY) THEN                    /* stack is empty */
       nxtCtx ← push_ctx(nxtCtx);                 /* push next context on stack */
     ELIF (TOP(active).eos ≥ ctx[nxtCtx].pre) THEN/* next context is descendant of current context */
       inner_loop_child(ctx[nxtCtx].pre);         /* process children of current context until next context */
       nxtCtx ← push_ctx(nxtCtx);                 /* push next context on stack */
     ELSE                                         /* next context is not descendant of current context */
       inner_loop_child(TOP(active).eos);         /* process all children of current context */
       POP(active);
                                                  /* no context node left */
   WHILE (active IS NOT EMPTY) DO                 /* finish all remaining active context nodes */
     inner_loop_child(TOP(active).eos);           /* process all remaining children of current context */
     POP(active);
   RETURN result;                                 /* return result */
 END

push_ctx (nxtCtx)
 BEGIN
   curPre ← ctx[nxtCtx].pre;
   eor ← curPre + doc[curPre].size;               /* end of result region */
   nxtChld ← curPre + 1;                          /* first child of current context */
   WHILE (ctx[nxtCtx].pre = curPre) DO            /* prune all duplicate context nodes */
     nxtCtx ← nxtCtx + 1;
   PUSH ⟨eor,nxtChld⟩ ON active;                  /* push current context on stack */
   RETURN nxtCtx;                                 /* return next context */
 END

inner_loop_child (eor)
 BEGIN
   nxtChld ← TOP(active).nxtChld;                 /* next child of current context */
   WHILE (nxtChld ≤ eor) DO                       /* iterate over all children in current region */
     APPEND ⟨nxtChld⟩ TO result;
     nxtChld ← nxtChld + doc[nxtChld].size + 1;   /* skip directly to next child */
   IF (nxtChld ≤ TOP(active).eor) DO              /* current context not yet finished */
     TOP(active).nxtChld ← nxtChld;               /* recall where to proceed */
   RETURN;
 END
```

Figure 4.8: Staircase join: `child` axis.

The staircase join algorithm of the `child` location step in Figure 4.8 has to cope with a special case. To return the result in document order it has to pause active context nodes during the evaluation of nested context nodes. A stack called *active* is used to store the

active nodes. All items on the stack, except the top item, which is the currently active node, represent paused context nodes. Like the other algorithms, the outer loop of the `child` step implementation iterates over the context nodes. The first branch of the conditional calls `push_ctx`. The helper function `push_ctx` only pushes the current context node to the stack, thus making it active, and removes duplicates (identical nodes). If there is already an active node the implementation calls `inner_loop_child` and either retrieves all result nodes (ELSE clause) or produces results until the next nested context node is reached and then pushes the nested context nodes on the stack (ELIF clause). The function `inner_loop_child` jumps from child to child until it reaches its given end of the region. If the end of the region is caused by the next context node the pre value of the next child is stored on the stack to remember where to proceed. After iterating over the complete context set an additional loop ensures that the paused context nodes are completed as well.

4.2.2 Loop-Lifted Child Step

The loop-lifted variants try to maintain as much as possible from their basic staircase join counterparts. One prerequisite is again the document order of the context node sequence. The main difference to the basic versions is that we need to retain duplicate nodes if they belong to different iterations.

The `child` axis requires only small modifications to lift the implementation, because overlap and the corresponding duplicate result nodes is restricted to the node identity. Figure 4.9 shows the processing of the `child` step, which is almost identical to the basic version. In the loop-lifted variant a result tuple is formed by the combination of the resulting node and the iter value of the context node. Instead of pruning the duplicate context nodes (in `push_ctx`), we now record the offset of the first and the last tuple (*fstIter* and *lstIter*). We enrich the active stack by two new fields that store the respective offsets. These offsets are used again during the generation of result nodes. A nested loop in `inner_loop_child` iterates over the context relation from *fstIter* to *lstIter* and adds the current result node together with all iterations. Everything else stays the same in the loop-lifted variant. The concepts of the basic staircase join like skipping and partitioning are still in use.

In the following example we use the algorithm in Figure 4.9 to evaluate a `child` step on the XML tree shown in Figure 4.10(a). The input are two item sequences, where the first context node sequence consists of the nodes *c* and *h* and the second one contains the node sequence (*g*, *h*, *l*). The *ctx* relation of the loop-lifted `child` step is listed in Figure 4.10(b). The evaluation starts with the context node *c* (pre value 2). In `push_ctx` *fstIter* and *lstIter* both point to the the first row of *ctx*. The generated result for the child node *d* thus consists of the tuple $\langle 1,3 \rangle$ (see also Figure 4.10(c)). After skipping, the evaluation proceeds at node *g* (6) and produces the result tuple $\langle 2,7 \rangle$ (node *h* in iteration 2). The nested node *h* appears in two iterations and the offsets of *fstIter* and *lstIter* therefore point to the third and fourth row of *ctx*, respectively. The inner loop in `inner_loop_child` iterates over these rows and the tuples $\langle 1,8 \rangle$ and $\langle 2,8 \rangle$ are appended to the result (node *i* in both iterations). Then node *g* becomes active again and produces result for the remaining two children ($\langle 2,9 \rangle$ and $\langle 2,10 \rangle$) skipping once more the subtree of node *j*.

4.2.3 Loop-Lifted Descendant Step

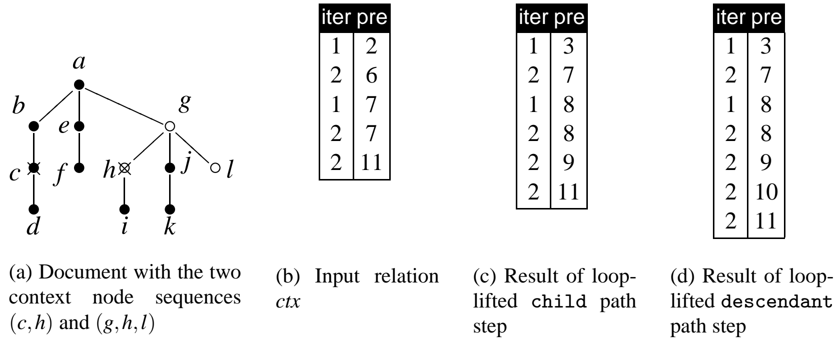
Loop-lifting the `descendant` axis modifies the basic staircase join algorithm much more than the lifting of the `child` location step. In comparison to the basic variant, it has to cope with nested context nodes belonging to different iterations. The difficulty becomes clearer if we apply a loop-lifted `descendant` step on the same context and document relations as the loop-lifted `child` step (see Figure 4.10(a) and 4.10(b)). Node *g* is active in iteration 2 and node *h* is active in iteration 1. Since we cannot prune node *h* anymore, both nodes have to be active at the same time. Our solution is to borrow the stack idea from the implementation of the `child` axis. The stack (*active*) then holds the active iter values as

```
ll_scj_child (doc : TABLE(pre,size), ctx : TABLE(iter,pre))
 BEGIN */
   ASSERT (doc.pre IS DENSE AND ASCENDING);          /* for positional lookup */
   ASSERT (ctx IS SORTED ON (pre));                   /* document order */
   result ← NEW TABLE(iter,pre);
   nxtCtx ← 0;
   lstCtx ← SIZE(ctx);
   active ← NEW STACK(eor,nxtChld,fstIter,lstIter);   /* stack of active context nodes */
   WHILE (nxtCtx ≤ lstCtx) DO                         /* iterate over all context nodes */
     IF (active IS EMPTY) THEN                        /* stack is empty */
       nxtCtx ← push_ctx(nxtCtx);                     /* push next context on stack */
     ELIF (TOP(active).eor ≥ ctx[nxtCtx].pre) THEN    /* next context is descendant of current context */
       inner_loop_child(ctx[nxtCtx].pre);             /* process children of current context until next context */
       nxtCtx ← push_ctx(nxtCtx);                     /* push next context on stack */
     ELSE                                             /* next context is not descendant of current context */
       inner_loop_child(TOP(active).eor);             /* process all children of current context */
       POP(active);
                                                      /* no context node left */
   WHILE (active IS NOT EMPTY) DO                     /* finish all remaining active context nodes */
     inner_loop_child(TOP(active).eor);               /* process all remaining children of current context */
     POP(active);
   RETURN result;                                     /* return result */
 END

push_ctx (nxtCtx)
 BEGIN
   curPre ← ctx[nxtCtx].pre;
   eor ← curPre + doc[curPre].size;                   /* end of result region */
   nxtChld ← curPre + 1;                              /* first child of current context */
   fstIter ← nxtCtx;                                  /* first appearance of current context node */
   WHILE (ctx[nxtCtx].pre = curPre) DO                /* prune all duplicate context nodes */
     nxtCtx ← nxtCtx + 1;
   lstIter ← nxtCtx − 1;                              /* last appearance of current context node */
   PUSH ⟨eor,nxtChld,fstIter,lstIter⟩ ON active;      /* push current context on stack */
   RETURN nxtCtx;                                     /* return next context */
 END

inner_loop_child (eor)
 BEGIN
   nxtChld ← TOP(active).nxtChld;                     /* next child of current context */
   fstIter ← TOP(active).fstIter;
   lstIter ← TOP(active).lstIter;
   WHILE (nxtChld ≤ eor) DO                           /* iterate over all children in current region */
     FOR i FROM fstIter TO lstIter DO                 /* iterate over all iters of current context */
       APPEND ⟨ctx[i].iter,nxtChld⟩ TO result;        /* append (iter,pre) to result */
     nxtChld ← nxtChld + doc[nxtChld].size + 1;       /* skip directly to next child */
   IF (nxtChld ≤ TOP(active).eor) DO                  /* current context not yet finished */
     TOP(active).nxtChld ← nxtChld;                   /* recall where to proceed */
   RETURN;
 END
```

Figure 4.9: Loop-lifted staircase join: `child` axis.

well as the last pre value where they are active. In comparison to the `child` step, not only the top item is active but all items on the stack are active at the same time.

The algorithm in Figure 4.11 still iterates over the context node sequence. In comparison to the basic staircase join version, it does not evaluate a complete subtree before consuming the next context node. Instead it consumes the next context node as soon the



iter	pre
1	2
2	6
1	7
2	7
2	11

iter	pre
1	3
2	7
1	8
2	8
2	9
2	11

iter	pre
1	3
2	7
1	8
2	8
2	9
2	10
2	11

(a) Document with the two context node sequences (c,h) and (g,h,l)

(b) Input relation *ctx*

(c) Result of loop-lifted `child` path step

(d) Result of loop-lifted `descendant` path step

Figure 4.10: Evaluation of different loop-lifted path steps starting from the *ctx* relation in 4.10(b). The context nodes of the first sequence (c,h) are marked with $\times$ and the nodes of the second sequence (g,h,l) are depicted with $\circ$.

pre value stored in the variable *cur_node*, that iterates over the document, is aligned with the context node. The nested conditional prunes all context nodes whose iterations are already active and adds the context nodes and its iteration value to the stack otherwise. So all duplicates within iterations are avoided. In the loop-lifted variant skipping is only possible if no context node is active (meaning the stack is empty). The first ELIF clause implements this skipping. The second ELIF clause as well as the ELSE clause are similar to the `child` step evaluation again. The ELIF clause calls the function `finish_region` that itself calls `inner_loop_desc`. The function `inner_loop_desc` scans the document *doc* in a given range and produces for each document node result tuples for all active iterations. Afterwards the variable *cur_node* is set to the last processed document node. The function `finish_region` additionally removes all items from the stack, which reached the end of their region. Clearing the stack, after the last context node is consumed, completes the loop-lifted `descendant` step.

The following example evaluates a loop-lifted `descendant` step on the document in Figure 4.10(a) and the context relation in Figure 4.10(b). The processing starts with the context node c (2) whose iter value (1) and the last node of the region (3) are pushed on the *active* stack. In the second iteration the third branch of the condition is chosen $(3 < 6)$ where the result tuple $\langle 1,3 \rangle$ is added to the result. Additionally the condition in `finish_region` becomes true and the only stack item is removed. In the third iteration the first condition is still false $(3 \neq 6)$. Since the stack is empty, the variable *cur_node* is set to 6. In the next iteration the first condition matches and the last descendant of node g as well as the iter value $(\langle 11,2 \rangle)$ are pushed on the stack. In the following iteration the last branch of the conditional is chosen for the first time. The descendants of node g are evaluated until the next context node is reached. Here only node h in iteration 2 is added to the result $(\langle 2,7 \rangle)$. The next iteration pushes a second item on the stack: the last descendant of node h and iter value 1. Then the first condition matches again. But node h in the second iteration is discarded because iteration 2 is already active. The evaluation proceeds and finishes the scan of the top stack item, where the document nodes in the range are produced for both active iterations $(\langle 1,8 \rangle, \langle 2,8 \rangle)$. After removing the top of the stack only the item representing node g remains active. In the following the tuples $\langle 2,9 \rangle$ and $\langle 2,10 \rangle$ are generated and node l is ignored. The complete context node relation is now consumed and the additional loop finishes the remaining active context nodes — here popping the last item completes the evaluation of the `descendant` step.

```
ll_scj_desc (doc : TABLE(pre, size), ctx : TABLE(iter, pre))
 BEGIN
   ASSERT (doc.pre IS DENSE AND ASCENDING);  /* for positional lookup */
   ASSERT (ctx IS SORTED ON (pre));          /* document order */
   result ← NEW TABLE(iter, pre);
   nxtCtx ← 0;
   lstCtx ← SIZE(ctx);
   nxt_node ← ctx[nxtCtx].pre;
   cur_node ← nxt_node;                      /* start scanning at first context node */
   active ← NEW STACK(eor, iter);            /* stack of active iterations */
   WHILE (nxtCtx ≤ lstCtx) DO                /* iterate over all context nodes */
     nxt_node ← ctx[nxtCtx].pre;
     IF (nxt_node = cur_node) THEN           /* context and document relation are aligned */
       IF (ctx[nxtCtx].iter NOT ON active) THEN  /* pruning of context nodes */
         eor ← cur_node + doc[cur_node].size;
         PUSH ⟨eor, ctx[nxtCtx].iter⟩ ON active;  /* add context node with region and new iteration */
       nxtCtx ← nxtCtx + 1;
     ELIF (active IS EMPTY) THEN             /* skip to next context node */
       cur_node ← nxt_node;
     ELIF (TOP(active).eor < nxt_node) THEN  /* next node is no descendant of the active node */
       cur_node ← finish_region(cur_node);   /* finish active region */
     ELSE                                    /* next node is descendant of the active node */
       inner_loop_desc(cur_node, nxt_node);  /* find all results til nxt_node */
       cur_node ← nxt_node;
   WHILE (active IS NOT EMPTY) DO            /* process all remaining active regions */
     cur_node ← finish_region(cur_node);
   RETURN result;
 END

finish_region (cur_node)
 BEGIN
   eor ← TOP(active).eor;
   inner_loop_desc(cur_node, eor);           /* find all results in the current region */
   cur_node ← eor;
   WHILE (TOP(active).eor = eor) DO          /* remove all iterations whose regions are processed */
     POP(active);
   RETURN cur_node;
 END

inner_loop_desc (first, last)
 BEGIN
   FOR pre FROM first + 1 TO last DO         /* skip self node; for every doc node in the range */
     FOREACH DISTINCT ⟨_, iter⟩ ON active DO /* and for every active iteration */
       APPEND ⟨iter, pre⟩ TO result;         /* add a result tuple */
 END
```

Figure 4.11: Loop-lifted staircase join: `descendant` axis.

4.2.4 Early Name and Kind Tests

Until now, we completely ignored any name and kind test and focused on the evaluation of the axis specifier. While this specifier may be already enough to answer certain queries, most XPath location steps use an additional name or kind test. These tests may retain all tuples generated by the axis specifier unchanged or may be highly selective. In the latter case, applying the name or kind test as a post-processing step seems to waste resources. An early name or kind test therefore helps to avoid extra evaluation and space consumption for certain queries. (Loop-lifted) staircase join behaves like any other relational join. In our query plans we can safely push selections (*e.g.*, name/kind tests) down through the staircase

join, to reduce the amount of tuples to process as early as possible. Duplicate result nodes in different iterations, whose name or kind has to be checked separately after the result generation, make another argument for early tests.

Because the staircase join requires the pre/size document relation for the skipping, our implementation uses a third input relation that keeps candidate nodes. The candidate node list stores the pre value of any node whose kind or name satisfies the test. It is sorted in document order like the other input relations. The candidate list is scanned together with the document in a merge-like synchronized sweep. As soon as a node satisfies the given axis specifier, a matching tuple in the candidate list confirms the match. If the candidate list does not contain the pre value the apparent result node is discarded. To avoid scanning (possibly large) parts of the candidate list, while these parts are just skipped in the document, we perform a binary search on the remaining nodes of the candidate list.

```
scj_desc (doc : TABLE(pre,size), ctx : TABLE(pre), cand : TABLE(pre))
BEGIN
  ASSERT (doc.pre IS DENSE AND ASCENDING);              /* for positional lookup */
  ASSERT (ctx IS SORTED ON (pre));                      /* document order */
  ASSERT (cand IS SORTED ON (pre));                     /* document order */
  result ← NEW TABLE(pre);
  nxtCtx ← 0;
  lstCtx ← SIZE(ctx);
  nxtCand ← 0;
  lstCand ← SIZE(cand);
  cur_node ← 0;
  WHILE (nxtCtx ≤ lstCtx AND nxtCand ≤ lstCand) DO      /* iterate over all context nodes */
    IF (cur_node ≤ ctx[nxtCtx].pre) THEN                /* pruning */
      cur_node ← ctx[nxtCtx].pre;                       /* skipping */
      nxtCand ← FIRST c ≥ nxtCand                       /* binary search */
                  WITH cand[c].pre ≥ cur_node;
      lst_node ← cur_node + doc[cur_node].size;
      cur_node ← cur_node + 1;                          /* omit self node */
      WHILE (cur_node ≤ lst_node AND                    /* collect all descendant nodes */
                 nxtCand ≤ lstCand) DO
        IF (cand[nxtCand].pre < cur_node) THEN          /* align candidate list */
          nxtCand ← nxtCand + 1;
        ELIF (cand[nxtCand].pre = cur_node) THEN        /* return matching tuple and proceed */
          APPEND ⟨cur_node⟩ TO result;
          cur_node ← cur_node + 1;
        ELSE                                            /* skip non matching node */
          cur_node ← cur_node + 1;
    nxtCtx ← nxtCtx + 1;
  RETURN result;
END
```

Figure 4.12: Staircase join: `descendant` axis with candidate list.

Figure 4.12 shows the basic staircase join of a `descendant` location step extended with candidate list processing. The skipping in the candidate list is imitated by a binary search. The inner loop of the staircase join furthermore aligns the current candidate node and confirms or prunes the descendant nodes. Similar extensions apply for all loop-lifted staircase join implementations.

The following example evaluates the XPath location step `descendant::text()` starting from the context sequence (c,g,h,l) (see also Figure 4.13(b)). The nodes in Figure 4.13(a) marked with $\times$ as well as the relation in Figure 4.13(c) illustrate the candidate nodes. The evaluation starts at node c and the first candidate node is node d (3). Inside the inner loop variable *cur_node* matches the candidate node d and becomes the first result tuple (depicted by the first row in Figure 4.13(d)). The next context node g becomes active

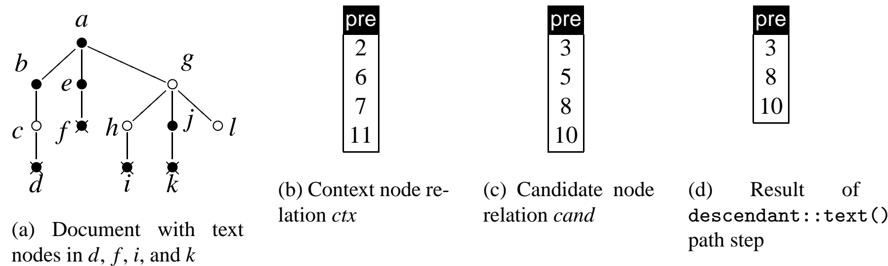


Figure 4.13: Evaluation of the XPath location step `descendant::text()` starting from the *ctx* relation in 4.13(b). The context nodes are marked with $\circ$ and the candidate nodes are depicted with $\times$.

and the binary search in the candidate relation skips node f and returns node i as new candidate (8). In the inner loop node h does not match the candidate node i and gets discarded. The next node matches i and is appended to the result. In the following iteration candidate node i has a smaller pre value than the current node. Therefore the next candidate node (k) becomes active and prunes node j in the next iteration. After adding node k to the result the current node is larger than the candidate node again and we increase variable *nxtCand*. Because all candidate nodes are now consumed, the evaluation stops and the evaluation of the `descendant::text()` step is complete.

The candidate list of the previous example contained all preorder ranks of text nodes. To retrieve such a kind list one kind in the kind column of the pre/size table can be selected. Because these selections would occur for every path step at runtime, we generate an index for every kind already during document shredding. These indices can be used as candidate lists for the kind tests. To support name tests the element index has a second column, which additionally stores the reference id from column ref of the pre/size document table. The generation of a candidate list for a name test then maps to the selection of a given reference in the element node index. For names with wildcards multiple references may match. An order preserving join then does the job.

4.2.5 Loop-Lifted Path Steps for Other Axes

In the previous section we only described the loop-lifting of the `child` and `descendant` axes. These are also the only axes used in the XMark benchmark, which we will use in the experiments in following section. But loop-lifting is possible for the other axes as well. In the following paragraphs we shortly sketch the ideas for the implementation of the remaining XPath axes and conclude this section with a short discussion of the drawbacks of the applied techniques.

self The `self` axis requires no document relation for the evaluation of the axis specifier, as pruning duplicate nodes within iterations works on the pre value alone and the name and kind tests can be processed by a merge-like scan over the context and candidate relation.

attribute Because we do not include the attributes in the pre/size relation but in a separate attribute relation, an `attribute` path step consists only of a join between the context sequence and the attribute relation on the foreign key column pre.

descendant-or-self The implementation of the `descendant-or-self` axis is almost identical to the `descendant` axis. The only modifications are minor shifts of the ranges to include the self nodes.

following-sibling An implementation for the `following-sibling` location step is similar to the one of the `child` axis. It also uses a stack of context nodes, where only the top item represents the active node c. In comparison to the `child` step, the first node is not the next node in the subtree, but the next sibling ($v_1.\mathsf{pre} = c.\mathsf{pre} + c.\mathsf{size} + 1 \ \wedge \ v_1.\mathsf{level} \geq c.\mathsf{level}$)

and the end of a region has to be determined by the level (v_i.level < c.level). The evaluation of the result nodes, however, applies the same skipping as a `child` step. A distinct list of active iterations within each stack item furthermore ensures the duplicate elimination.

following For the basic staircase join the context nodes sequence of the `following` step can be pruned to a single node. In the loop-lifted variant each iteration can be pruned to one context node. During the sequential scan over the document each context node either can be pruned because its iteration is already active or the list of active iter values (similar to the active stack of the `descendant` step) is extended by the new value. Each matching document node is then emitted for all values in the active iter list.

ancestor and ancestor-or-self The two axes `ancestor` and `ancestor-or-self` can be implemented by combining the ideas of the `child` and the `descendant` steps. While the latter two keep a stack for active context nodes, the `ancestor` axis keeps a stack with the currently active path to the root. It uses the same skipping as the basic staircase join implementation and emits the active path together with the iter value as soon as a context node is reached. To maintain document order every tuple is added into a list according to its level. Duplicate result nodes are avoided by using an additional list of highest pre values per iteration to discard nodes on the currently active path stack. The multiple result lists (one per level), which are sorted on document order, are merged afterwards to return the overall result in document order as well.

parent The implementation of the `parent` axis uses the same ideas as the `ancestor` step, but adds only the parent node on the active path stack to the result lists.

preceding-sibling For the `preceding-sibling` location step the `ancestor` step implementation also plays an important role. It uses an additional list of preceding-sibling nodes for each item on the active path stack to collect the possible result nodes. A second list of active context nodes is then used to generate the results for all active iterations in document order.

preceding The context nodes of the `preceding` axis can be pruned to one context node per iteration similar to the `following` axis. For these remaining context nodes we can retrieve the `ancestor` nodes that can be used as second candidate list. In comparison to the candidate list for name or kind tests, this list is used to prune only the `ancestor` nodes during the scan over the document.

The implementations of the axes can be grouped into three categories. The first group requires no special staircase join operator and retrieves its result with ordinary database operations. The two axes `self` and `attribute`, which form that group do not rely on the tree structure of the XML document and therefore have no use for concepts like skipping. The second category consists of the forward axes and the third is put together of the reverse axes. Both groups make use of the staircase join concepts pruning, partitioning, and skipping. For the reverse axes, however, processing requires more internal state keeping and intermediate results (*e.g.*, level lists to support document order). But the main cost factor in comparison to the iterative application of the basic staircase join is the sorting overhead required to match the physical order.[2] These are the sorting costs for the creation of the input in item order and the output in iter|item order. The next optimization in Section 4.3 will describe how this sorting overhead can be avoided in a list of path steps. Because most queries start from a single context node, the item sorting for the first path step may be cheap. What remains is the need to produce the output of the last path step sorted on iter and then on item. One solution to avoid the sorting is to use a separate result list for each iteration in the last path step. Because result nodes are generated in document order, the lists are also sorted in document order. The concatenation then returns the result sorted first on iter and secondly on item.

[2]The performance differences for avoiding multiple scans of the document relation are ignored here.

4.3 Avoiding the iter|pos|item|kind Interface

To support heterogeneous sequences our item encoding copes with polymorphic types. In most cases, however, the iter|pos|item|kind representation encodes sequences with homomorphic types. Very often the values of these sequences are added to the value relations only to be looked up again by the following operation. As we could store such values directly in a single column, the indirection is certainly wasteful.

In our compilation scheme we addressed this problem by introducing additional *interfaces*. These new interfaces allow to return intermediate results using representations other than the the iter|pos|item|kind relation. The first different representation was already sketched in the motivating example. The *value interface* avoids adding values to their value relations if they are immediately required by the next construct. It can be used whenever the returning intermediate result has a monomorphic static type. The decision whether the intermediate result returns its values using the value interface or the normal iter|pos|item|kind representation is solved at compile time and the calling construct adapts its MIL code accordingly.

In Query Q_{15} the value interface is used for both arguments of the `contains` function:

```
for $a in doc("fairy_tailes.xml")//story return
where contains ("gold", string(data($a/text()))) .
return $a/@name
```
(Q_{15})

As `contains` directly works on the strings, both intermediate results return an iter|pos|item$_{string}$ relation. The more `story` nodes the path step in Query Q_{15} returns the more we benefit from the modified compilation, which avoids the insertion and its duplicate elimination.

The second interface uses an iter|item|kind representation and is used only between directly following path steps. While it does not seem to be different from the normal interface, it exploits the information obtained in the previous two sections (4.1 and 4.2). The result of the first path step does not have to be sorted on iter and item, since the second path step wants its input sorted in document order item only. We thus omit two sort operations without any drawbacks.

If we reconstruct XML document *Doc* in XQuery (see Query Q_{16}) we have to keep the XQuery semantics in mind.

```
<a><b>c</b><d><e/><f/></d><g a="42"/></a>
```
(Q_{16})

These require each node in the element body to be a subtree copy of the element content. In Query Q_{15} we create 13 element nodes, 3 text nodes, and 3 attribute nodes to recursively built a result with 8 nodes. The evaluation cost and the storage overhead explode with a growing number of nodes and increasing nesting depth.

The *node interface* speeds up the nested node construction. It makes use of the fact, that we do not need the constructed element content after the subtree copy anymore, if it is not referenced elsewhere. In such a case, the mapping can act similar to the value interface and store the constructed nodes in an intermediate result instead of the transient document node container. These nodes then can be used as subtree copies and require no explicit copying. A recursive XML tree construction then creates each node exactly once and additionally avoids the path steps to retrieve the subtree nodes, since these nodes are already in the intermediate result.

In addition to the changes in the node constructors, which cope with both, the normal and the node interface, we added a separate mapping for the sequence constructor, which combines multiple subtrees and attributes. The relations this node interface uses, match the pre and attr relations of the transient document node container extended with the iteration information. All other values like, *e.g.*, element names or text values are already stored in the transient document node container Δ, because they are not changed anymore.

iter	pre	size	level	ref	kind	frag	cont
1	1@0	0	0	0@0	elem	1	Δ
1	2@0	0	0	1@0	elem	1	Δ

(a) Input sequence using the node interface (nodes `e` and `f`)

iter	pre	size	level	ref	kind	frag	cont
1	1@0	2	0	2@0	elem	1	Δ
1	2@0	0	1	0@0	elem	1	Δ
1	3@0	0	1	1@0	elem	1	Δ

(b) Result of element construction using the node interface (node `d`)

Figure 4.14: Content and result of element construction using the node interface

Figure 4.14(a) shows the input relation of the element construction of node `d` from Query Q_{16} and Figure 4.14(b) its returning representation. After compiling the content expression the element construction skips the code generation of the content nodes in Rule ELEM. Instead only the level of all content nodes is increased by one. The MIL code creation for the root nodes remains the same. If the result expression is compiled using the node interface like, *e.g.*, node `d` the node insertion phase of the ELEM rule has to be modified. The main difference is that the new tree representation is not added to the transient document node container but returned as result relation. Furthermore, both pre and frag values start at 1 and the result relation $nodes_{new}$ also retains the column iter. For node `d` the intermediate result is displayed in Figure 4.14(b). Note that the element names are already stored in the transient document node container (see line (11) in Rule ELEM). For the attributes a similar modification is performed. Two attr relations that are both extended with the column iter are used as intermediate storage. The reason for the two relations is the distinction between root attributes and content attributes.

4.4 Join Recognition

At the beginning of this section we mentioned the performance problems of the compilation scheme without join recognition. The reason is the naive compilation of nested XQuery `for` expressions, which lead to the computation of Cartesian products. The query

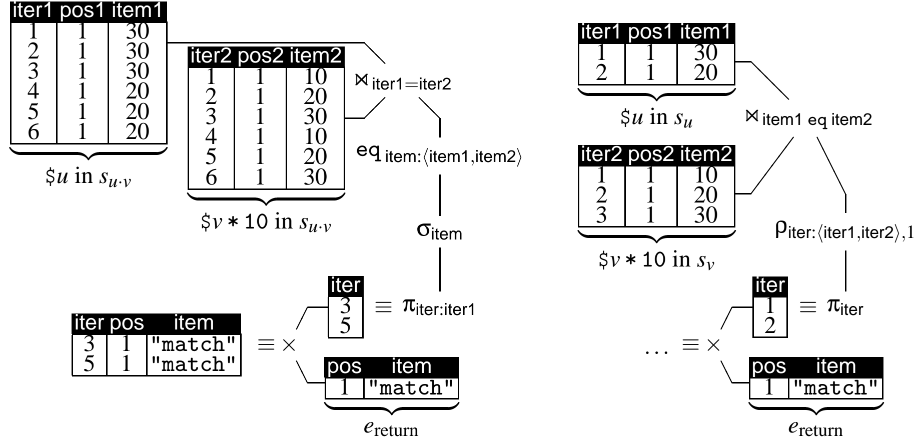
$$s_0 \left\{ \begin{array}{l} \texttt{for } \$u \texttt{ in (30, 20)} \\ s_u \left\{ \begin{array}{l} \texttt{for } \$v \texttt{ in (1, 2, 3)} \\ s_{u \cdot v} \left\{ \begin{array}{l} \texttt{where } \$u \texttt{ eq } \$v \texttt{ * 10} \\ \texttt{return "match"} \end{array} \right. \end{array} \right. \end{array} \right. \qquad (Q_{17})$$

is a simplified but representative instance of such a nested `for` expression. The compilation rules given in Section 3 require both argument expressions of the `eq` operator $\$u$ and $\$v * 10$ to be represented with respect to scope $s_{u \cdot v}$ (see Figure 4.15(a)). Consequently, the intermediate result $\$v * 10$ is compiled in dependence of scope s_u and is thus explicitly represented for each binding of $\$u$. For the resulting expression `"match"`, however, only the loop relation generated from the matching iterations is important. The observation that both arguments of the equality expression are *independent* of each other enables us to avoid the nested evaluation. Instead we can split up the compilation of the comparison input arguments:

```
for $u in (30, 20)      for $v in (1, 2, 3)
   return $u               return $v * 10
```

The application of a *relational join* between the two *independent* item sequences then provides the same loop relation for the result expression `"match"` (see Figure 4.15(b)) as the loop-lifted version that builds a Cartesian product.

In the following we propose a join pattern that detects a large subset of the XQuery joins, including those in the XMark benchmark set [25] or in Appendix H.1 ("Joins") of the W3C



(a) Relational evaluation of XQuery example Q_{17}. Evaluation of predicate `eq` requires both operands ($\$u$ and $\$v * 10$) to be represented loop-lifted with respect to the inner `for` loop. iter values that satisfy the `eq` predicate form the loop relation for the loop-lifted evaluation of the `return` clause.

(b) Join plan for the evaluation of example Q_{17}. Both operands are computed independently. The join result ultimately serves as relation loop to compile the `return` part.

Figure 4.15: Simplified evaluation of Query Q_{17} without (a) and with join recognition (b) applied.

XQuery Working Draft [5]. After checking the conditions necessary for an independent evaluation we describe the mapping to algebraic join plans in Section 4.4.2. An inference rule similar to the ones in Chapter 3 formalizes our translation. Because most XQuery joins compare sequences of items instead of single values, we describe an extended join pattern and the respective adoption of the translation in Section 4.4.3.

4.4.1 Join Pattern

The starting point of the join detection is a normalized XQuery Core expression. It is the basis to recognize the following pattern at arbitrary nesting depth:

$$\texttt{for } \$v \texttt{ in } e_{\text{in}} \quad \texttt{return if } (p(e_1, e_2)) \texttt{ then } e_{\text{return}} \texttt{ else () .} \qquad (P_1)$$

This subexpression qualifies for an XQuery join, if

(i) variable $\$v$ does not appear free in e_1[3],

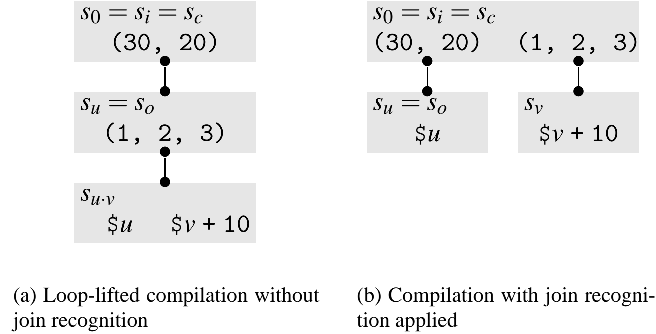
(ii) variables occurring free in e_2 and e_{in} are bound in any enclosing scope, except for the scope that *directly* encloses P_1, and

(iii) predicate p is supported by the theta-join implementation of the relational back-end (*i.e.*, typically, p will be `eq`, `lt`, ..., or one of the XQuery general comparison operators with existential semantics like =, <; see Section 4.4.3).

The normalized XQuery Core expression of Query Q_{17} looks like Query Q_{18}

```
s_0 { for $u in (30, 20)
s_u {   for $v in (1, 2, 3)
s_u·v {   return if ($u eq $v * 10)
                then "match" else ()
```
(Q_{18})

[3]The roles of e_1, e_2 may be arbitrarily swapped.



(a) Loop-lifted compilation without join recognition

(b) Compilation with join recognition applied

Figure 4.16: Tree of scopes for Query Q_{18}, which shows the nesting of the scopes for the loop-lifted translation (a) and for the compilation with join recognition applied (b). For the latter case the arguments of the operator `eq` are shifted into higher, less expanded scopes.

whose `where` clause has been replaced by an `if` expression. The join pattern

```
for $v in (1, 2, 3)  return if ($u eq $v * 10) then "match" else ()
```

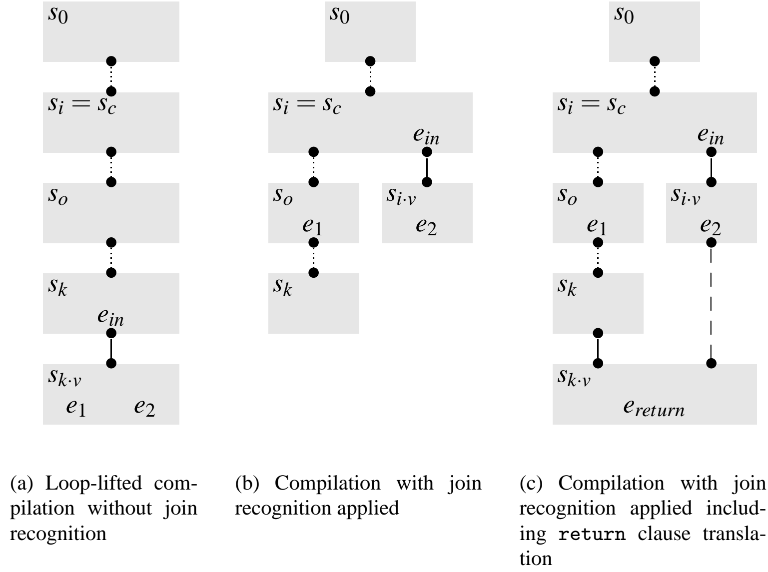
in Query Q_{18} is clearly visible. The item sequence `(1, 2, 3)` matches e_{in}, `eq` is the join predicate, `$u` is e_1, `$v * 10` is e_2, and `"match"` corresponds to e_{return}.

The `eq` operator in Query Q_{18} can be evaluated in MonetDB and thus satisfies the XQuery join condition *(iii)*. The other two conditions require a check of the variable occurrences in the three XQuery expressions e_1, e_2, and e_{in}. In Query Q_{18} the variable `$v` appears free in e_2 only and thus matches the first condition. The only variable form the directly enclosing scope `$u` appears free in e_1 only, which fulfills the last remaining requirement (condition *(ii)*) for a valid join pattern.

4.4.2 Join Translation

Recognizing the join pattern and checking the independent execution of both join arguments is the first important step to compile an XQuery join efficiently. The join translation exploits this independence to compile the input arguments of the theta-join in the least expanded scope thus minimizing the sizes of the input relations. The compilation is described by means of an inference rule that, in comparison to the earlier inference rules in Chapter 3, has to cope with multiple scopes. The relationship between the scopes and the mapping can be explained using a tree of scopes. In this tree, scopes are represented by nodes (or frames), whose connecting edges depict `for` expressions. Figure 4.16(a) shows the tree of scopes for Query Q_{18}, which is evaluated with a Cartesian product. Both expressions of the join pattern comparison are evaluated in the inner-most scope $s_{u \cdot v}$. The join-aware compilation on the other hand splits up the tree in such a way that both arguments of the theta-join are evaluated in independent scopes. Figure 4.16(b) illustrates the modified tree of scopes for Query Q_{18} where the join recognition is applied. The `for` loop binding of e_{in} is compiled in dependence of scope s_0 and the second join argument e_2 in the therein nested scope s_v. In comparison, the first argument of the theta-join is evaluated in scope s_u, since this scope introduces the inner-most free variable of the expression e_1. The results are the two input relations visible in Figure 4.15(b).

In the above example, both input arguments of the theta-join only depend on variables in the directly enclosing scopes. In the more general case, however, they may depend on variables in higher scopes. Figure 4.17(a) shows the tree of scopes for the general case where the named scope s_i depicts the inner-most scope that binds a variable, which appears free in e_2 or e_{in}. Because either e_2 or e_{in} need to access this variable binding, the inner `for` loop has to be evaluated in dependence of scope s_i. The same applies for e_1,



(a) Loop-lifted compilation without join recognition

(b) Compilation with join recognition applied

(c) Compilation with join recognition applied including `return` clause translation

Figure 4.17: Tree of scopes for the general join pattern. The nesting of the scopes for the loop-lifted translation compiles everything in the inner-most scope $s_{k \cdot v}$ (a). The join recognition breaks up the arguments of the `eq` operator and shifts them to the highest (outer-most) scope possible (b). The outer of the both scopes s_i and s_o determines the common scope s_c, which is used to link both input relations of the theta-join. The `return` clause is compiled in dependence of the current scope s_k and the both argument scopes (c).

which for the same reason has to be evaluated in dependence of scope s_o. The join-aware compilation exploits these dependencies and compiles the expression e_1 in scope s_o and `for $`v `in` e_{in} `return` e_2 in scope s_i (see Figure 4.17(b)). Because both scopes may depend on a common subset of scopes, we furthermore determine a common scope s_c, that corresponds to the outer scope of the scopes s_i and s_o[4]. The common scope s_c is necessary to link the two theta-join arguments. Otherwise both expressions are treated as if they are completely independent, which leads to the duplication of the surrounding scopes (s_0-s_c). For Query Q_{18} scope s_i and scope s_c is the outer-most scope s_0. Scope s_o matches scope s_u because e_1 references variable $\$u$, which is introduced in this scope (see Figure 4.15(b)).

The inference Rule JOIN in the following refers to these three named scopes. In order to apply the relational join, its arguments need to be mapped to their common scope s_c. The respective map relations correspond to joins along the scope hierarchy, indicated by the ellipsis in line (8) and (12). Furthermore w.l.o.g. we assume scope s_o to be nested in scope s_i. The compilation of an XQuery join starts with the mapping of the first theta-join argument using the loop relation as well as the variable environment Γ of scope s_o (line (2) of Rule JOIN). The `for` loop of the join pattern is compiled in dependence of scope s_i using the second theta-join argument as `return` clause (see lines (3) to (7)). Instead of mapping back the result, like in Rule FOR, the intermediate results are extended with their values and the outer column of the common scope s_c. While this may be a concatenation of several map relations for the first argument e_1 (from scope s_c to scope s_o), it is only the last map relation for the second argument e_2 ($\mathsf{map}_{(i,v)}$). The outer column of the common scope s_c links the two theta-join arguments to avoid result tuples in non-matching iterations. (see line (11) of Rule JOIN). Without the join predicate on the outer columns, the loop-lifting

[4]In the following, we assume w.l.o.g. scope s_i to be the common scope s_c.

from scope s_0 to scope s_c would be applied twice (for each theta-join input argument once).

$$
\begin{array}{ll}
(1) & e_1 :: kind \\
(2) & \Gamma_o; \mathsf{map}_o; \Delta \vdash e_1 \Rightarrow (q_1, \Delta_1) \\
(3) & \Gamma_i; \mathsf{map}_i; \Delta_1 \vdash e_{in} \mapsto (q_{in}, \Delta_2) \\
(4) & ext_{in} \equiv \rho_{inner:\langle iter,pos\rangle,1}(q_{in}) \\
(5) & q_v \equiv \boxed{\frac{pos}{1}} \times \pi_{iter:inner,item,kind}(ext_{in}) \\
(6) & \mathsf{map}_{(i,v)} \equiv \pi_{outer:iter,inner}(ext_{in}) \\
(7) & \left\{\forall \$v_n \in \Gamma_i : \$v_n \mapsto \pi_{iter:inner,pos,item,kind}\left(q_n \bowtie_{iter=outer} \mathsf{map}_{(i,v)}\right)\right\} + \{\$v \mapsto q_v\}; \\
 & \mathsf{map}_{(i,v)}; \Delta_2 \vdash e_2 \mapsto (q_2, \Delta_3) \\
(8) & \mathsf{map}_{(c,o)} \equiv \pi_{outer,inner}\left(\mathsf{map}_{(c,c+1)} \bowtie \cdots \bowtie \mathsf{map}_{(o-1,o)}\right) \\
(9) & q_{1\cdot extended} \equiv \pi_{iter_1:iter,value_1:value,outer_1:outer}\left(map_{(c,o)} \bowtie_{inner=iter} q_1 \bowtie_{item=ref} \Delta_3[kind]\right) \\
(10) & q_{2\cdot extended} \equiv \pi_{iter_2:iter,value_2:value,outer_2:outer}\left(map_{(i,v)} \bowtie_{inner=iter} q_2 \bowtie_{item=ref} \Delta_3[kind]\right) \\
(11) & q_{join} \equiv \pi_{iter_1,iter_2}\left(q_{1\cdot extended} \bowtie_{p(value_1,value_2)\wedge outer_1=outer_2} q_{2\cdot extended}\right) \\
(12) & \mathsf{map}_{(o,k)} \equiv \pi_{outer,inner}\left(\mathsf{map}_{(o,o+1)} \bowtie \cdots \bowtie \mathsf{map}_{(k-1,k)}\right) \\
(13) & q_{join\cdot mapped} \equiv \pi_{iter_1:inner,iter_2}\left(\mathsf{map}_{(o,k)} \bowtie_{outer=iter_1} q_{join}\right) \\
(14) & ext_{join} \equiv \rho_{inner:\langle iter_1,iter_2\rangle,1}\left(q_{join\cdot mapped}\right) \\
(15) & \mathsf{map}_{(k,k+1)} \equiv \pi_{outer:iter_1,inner}(ext_{join}) \\
(16) & q_{v\cdot mapped} \equiv \pi_{iter:inner,pos,item,kind}\left(q_v \bowtie_{iter=iter_2} ext_{join}\right) \\
(17) & \left\{\forall \$v_n \in \Gamma_k : \$v_n \mapsto \pi_{iter:inner,pos,item,kind}\left(q_n \bowtie_{iter=outer} \mathsf{map}_{(k,k+1)}\right)\right\} \\
 & + \left\{\$v \mapsto q_{v\cdot mapped}\right\}; \mathsf{map}_{(k,k+1)}; \Delta_3 \vdash e_{return} \mapsto (q_{return}, \Delta_4) \\
\hline
\end{array}
$$

$$
\begin{array}{c}
\Gamma_k; \mathsf{map}_k; \Delta \vdash \texttt{for } \$v \texttt{ in } e_{in} \texttt{ return if } p(e_1, e_2) \texttt{ then } e_{return} \texttt{ else ()} \mapsto \\
\left(\pi_{iter:outer,pos:pos_1,item,kind}\left(\rho_{pos_1:\langle iter,pos\rangle/outer,1}\left(q_{return} \bowtie_{iter=inner} \mathsf{map}_{(k,k+1)}\right)\right), \Delta_4\right)
\end{array}
\quad \text{(JOIN)}
$$

Query Q_{19} probably demonstrates the need for this additional predicate best:

s_0 { s_t { $s_{t\cdot u}$ { $s_{t\cdot u\cdot v}$ { (Q_{19})

```
for $t in (10, 10)
  for $u in (30, 20)
    for $v in (1, 2, 3)
      where $u eq $t * $v
      return "match"
```

Query Q_{19} contains a relational join and has the same XQuery join pattern as Q_{17} and Q_{18}. In contrast to the former two examples, it iterates over $\$t$ thus creating a four-item sequence as result. The scope s_o required for the expression e_1 is scope $s_{t\cdot u}$ and the scope of the pattern `for` loop s_i as well as the common scope s_c correspond to scope s_t. For the first join argument $\$u$ the theta-join input relation is the four-item sequence shown in Figure 4.18(a) and the second join argument ($\$t$ `*` $\$v$) is displayed in Figure 4.18(b). Because both relations are loop-lifted through scope s_t, we have to add the additional join predicate on the outer columns. Otherwise we would apply the outer-most `for` loop twice (resulting in 8 matches).

If the common scope s_c matches the outer-most scope s_0, like in Query Q_{18}, the outer relation of scope s_0 contains only the value 1 and the join predicate on the outer columns is always true (see Figure 4.19). Therefore the comparisons of the outer columns as well as the mapping of these columns can be skipped completely.

With the evaluation of the theta-join in line (11) of Rule JOIN, the matching iterations are in scope s_o (iter_1) and scope $s_{i\cdot v}$ (iter_2). They need to be mapped to the current scope s_k

iter_1	values_1	outer_1
1	30	1
2	20	1
3	30	2
4	20	2

(a) $\$u$ in scope $s_{t \cdot u}$ ($q_{1 \cdot extended}$)

iter_2	values_2	outer_2
1	10	1
2	20	1
3	30	1
4	10	2
5	20	2
6	30	2

(b) `$t * $v` in scope $s_{t \cdot v}$ ($q_{2 \cdot extended}$)

iter_1	iter_2
1	3
2	2
3	6
4	5

(c) Result of theta-join in scope $s_{t \cdot v}$ (q_{join})

Figure 4.18: Input and result relations of theta-join in Query Q_{19}

iter_1	values_1	outer_1
1	30	1
2	20	1

(a) $\$u$ in scope s_u ($q_{1 \cdot extended}$)

iter_2	values_2	outer_2
1	10	1
2	20	1
3	30	1

(b) `$t * $v` in scope s_v ($q_{2 \cdot extended}$)

iter_1	iter_2
1	3
2	2

(c) Result of theta-join in scope s_u (q_{join})

Figure 4.19: Input and result relations of theta-join in Query Q_{18}

because the `return` clause e_{return} may use variables introduced in this scope and the result of the join pattern is expected to be in scope s_k as well. The first step in the compilation is the creation of the map relation, which maps from scope s_o to scope s_k. Extending the theta-join result with the map relation ($map_{(o,k)}$) results in a relation ($q_{join \cdot mapped}$) that stores for each iteration in scope s_k (iter_1) the offsets of the tuples in the original inner-most `for` loop (iter_2), which are retained by the conditional (theta-join). The following row numbering that uses the two columns as sorting criterion prepares the mapping from scope s_k to $s_{k \cdot v}$ (see Figure 4.17(c)). The remaining compilation of the `return` clause e_{return} as well as the backmapping proceeds like every regular `for` loop compilation. The only difference is the transformation of the variable $\$v$, which is mapped from scope $s_{i \cdot v}$ using the iter_2 column of the extended join result (ext_{join}).

The theta-join results of Query Q_{18} and Query Q_{19} are already in scope s_k, which matches scope s_u and $s_{t \cdot u}$, respectively. The row numberings generate new dense inner columns, which, together with the columns iter_1 as outer columns, from the map relations to the scope $s_{k \cdot v}$. The variables $\$v$ are mapped to scope $s_{k \cdot v}$ by joining their representation with the iter_2 columns of the extended join results. In scope $s_{k \cdot v}$ of Query Q_{18} variable $\$v$ stores the value 3 in the first iteration and the value 2 in iteration 2.

4.4.3 Existential Semantics

General comparisons in XQuery like, *e.g.*, =, <, and >= are existentially quantified comparisons that may be applied to operand sequences of any length. The result of such a comparison is true if there exists at least one pair of values, which produces a match using the corresponding value comparison operator. The same applies for the quantified expression `some`. During the XQuery Core normalization all quantified expressions are translated into conditionals, nested `for` expressions, and value comparisons (depicted in the normalization of Query Q_{20} to Q_{21}).

```
for $u in (30, 20)
  for $v in (1, 2, 3)
    where $u = $v * 10
    return "match"
```

(Q_{20})

Query Q_{20} returns the same result as the Queries Q_{17} and Q_{18}, but uses a general comparison (namely = instead of eq). With static type checking, all three queries are normalized to Query Q_{18}. If we however ignore the static type checking Query Q_{20} gets normalized to Query Q_{21}:

```
for $u in (30, 20) return
  for $v in (1, 2, 3) return
    if (empty (for $tmp1 in $u return
          if (empty (for $tmp2 in $v * 10 return
                if ($tmp1 eq $tmp2)
                then 1
                else ()))
          then ()
          else 1))
    then ()
    else "match"
```

. (Q_{21})

The two additional for loops over tmp_1 and tmp_2 ensure that the value comparison in the inner-most scope only works on single values instead of sequences. The additional empty function calls are used to restrict the evaluation of the return expression "match" to the sequences, which have at least one matching value. The general patterns for an XQuery join with existential semantics (P_2) is almost identical:

```
for $v in e_in return
   if (empty (for $tmp1 in e1 return
                 if (empty (for $tmp2 in e2 return
                               if (p(tmp1,tmp2))
                               then 1
                               else ()))
                 then ()
                 else 1))
   then ()
   else e_return
```

. (P_2)

Similar to the simple join pattern P_1, four subexpressions (e_{in}, e_1, e_2, and e_{return}) and the join comparison p can be extracted . The three conditions from the simple pattern have to be fulfilled for this extended pattern as well. XQuery joins, where one argument contains a list of values and the other a single value, are matched by additional, but similar, patterns. All these pattern, however, share the same translation. The existential semantics effect only the result generation of the theta-join, which produces duplicate result tuples if more than one match per sequence occurs. An additional duplicate elimination step during or after the theta-join therefore has to ensure the correctness of the result.

Query Q_{22} is similar to the Queries Q_{17}, Q_{18}, Q_{20}, and Q_{21} and only differs in the first comparison argument that consists of a list of values instead of a single value ((20, $u)):

```
for $u in (30, 20)
  for $v in (1, 2, 3)
    where (20, $u) = $v * 10
    return "match"
```

. (Q_{22})

$iter_1$	$values_1$
1	20
1	30
2	20
2	20

(a) (20, $u)

$iter_2$	$values_2$
1	10
2	20
3	30

(b) $t * $v

$iter_1$	$iter_2$
1	2
1	3
2	2
2	2

(c) Theta-join result without duplicate elimination

$iter_1$	$iter_2$
1	2
1	3
2	2

(d) Theta-join result with duplicate elimination

Figure 4.20: Input and result relations of theta-join in Query Q_{22}

$iter_1$	$values_1$
1	30
2	20

(a) Pruned (20, $u)

$iter_2$	$values_2$
1	10
2	20
3	30

(b) Pruned $t * $v

$iter_1$	$iter_2$
1	1
1	2
2	1

(c) Theta-join result

Figure 4.21: Pruned input and result relations of inequality theta-join (replaced = in Query Q_{22} by <)

Figure 4.20 shows both input arguments of the theta-join as well as the result with and without duplicate elimination applied. Because the common scope s_c is the outer-most scope s_0, the outer columns and the corresponding join predicate are omitted.

since the duplicate elimination can become expensive with increasing document sizes, we extended our join translation with more intelligent duplicate elimination operations. For the equality join, we added a dynamic evaluation strategy, which samples the theta-join input relation and decides at runtime between two solutions. The first one uses a `HashJoin` implementation with a duplicate elimination and sorting step (to re-establish the order awareness). The second strategy is a `NestedLoopJoin`, which preserves the physical order and prunes the duplicates during evaluation.

Theta-joins with inequality predicates, which have to cope with existential semantics, can even prune the duplicates before the evaluation of the join. For both theta-join input arguments, depending on the predicate p, the maximum and accordingly the minimum value grouped by iterations is determined. The theta-join then only produces one tuple per combination of iterations at most.

If we replace the = operator in Query Q_{22} by a > operator, the input relations stay the same as depicted in Figure 4.20(a) and 4.20(b). The pruning step for the inequality operator > however looks up the maximum values grouped by iteration in the first argument and the minimum values (again per iteration) in the second theta-join argument. Figure 4.21 shows the pruned input as well as the output of the inequality join. The first argument is reduced to the value 30 in iteration 1 and 20 in iteration 2. The second argument stays unchanged since there already was only value per iteration.

Chapter 5

Experiments

After the explanation of our compilation scheme and description of the various optimizations it is time to deliver the hard facts. We therefore show the benefit of the optimizations described in Sections 5.1–5.4. Because the main focus of this work was to build a scalable XQuery system, we demonstrate the performance of our implementation on XML documents up to 11 GB in Section 5.5. We complete our experimental evaluation by comparing the implementation of Pathfinder/MonetDB (MonetDB/XQuery) [22] to other approaches.

Our experimentation platform was a 1.6 GHz AMD Opteron 242 (1 MB L2 cache) system with 8 GB RAM and a RAID-5 disk subsystem (3ware 7810, configured with eight 250 GB IDE disks of 7200 RPM). The operating system was Linux 2.6.9, using a 64-bit address space. In the experiments, we focused on the XMark benchmark [25], which consists of 20 queries and is the most frequently used benchmark to evaluate XQuery efficiency and scalability. The first column of Table 5.1 shows the tasks of the different queries (see also the complete XMark query set in Appendix A). With the XML generator from the XMark project, *XMLgen*, we generated documents of sizes 11 MB, 110 MB, 1 GB, and 11 GB (scaling factor 0.1 to 100).

For the XMark test set the query compilation times vary between 60 and 100 ms. A 11 MB XML document instance is loaded in 835 ms and its query result is serialized in less than 50 ms (except for query Q10 that requires 690 ms as it returns a large part of the document). For all following experiments we excluded query parsing, document loading, and serialization times thus measuring only the pure query evaluation time. Each query was evaluated 5 times in a row and the optimum was chosen. We only observed a small variance in the measured values, and all measurements were easily reproducible.

5.1 Order Awareness

We used two different versions of the Pathfinder compiler to generate MIL code for the XMark queries, which was then evaluated on a 110 MB XML document. The first version uses explicit sorting to support the row numbering operators (`mark` and `mark_grp`) and accesses all available algorithms in MonetDB. The second version that applies the optimization described in Section 4.1 uses only the order preserving operators and avoids sorting as much as possible. Figure 5.1 shows both versions normalized to the evaluation times of the non order preserving version. For all queries the order aware version performs better. In most cases, we observe an improvement of more than factor 2. Query Q1 evaluates a particularly simple query, which contains only a small number of order operations. It is quite remarkable that the overhead of the sorting operations is about 40% of the overall evaluation time.

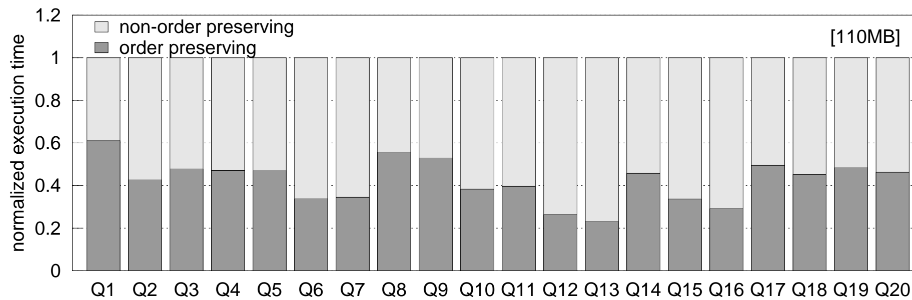


Figure 5.1: Benefits of order preserving operators. The evaluation times are normalized to the non-order preserving implementation. The order preserving variant shows an average performance improvement of factor 2.3.

5.2 Loop-Lifted Path Steps

Figure 5.2 shows the effect of using loop-lifted staircase join. We again used the 110 MB XML document and normalized the evaluation times to the measured times for the original version (see broad lightgray bars). While the original version calls the basic staircase join for each iteration separately, the loop-lifted staircase join evaluates a path in one sequential pass over the pre|size table for multiple sequences of context nodes in one go. In the XMark benchmark only the `child` and the `descendant` steps are relevant.

The leftmost narrow bar of each query shows the speedup achieved by the loop-lifting of the `child` step alone. The performance improvements are already of a factor 2, if we ignore the queries Q6 and Q7, which do not contain `child` steps. The queries Q8–Q12 furthermore spend a large part of their evaluation time on an XQuery join and Q14 requires most of its execution time in the evaluation of a full-text search and its preparation. Query Q1 spends most of its time in path steps in the outer-most scope with only a single iteration and the remaining part in nested path steps. The overhead for evaluating the lifted version in the outer-most scope compensates the performance gain in the nested scopes. Queries Q15 and Q16 are an even better example. Both queries evaluate a particularly long path expression of 13 steps. While query Q15 does evaluate all steps outside any `for` loop, query Q16 uses all result tuples of the fourth path step as one-item context node sequences for the remaining steps. The corresponding performance results are not surprising. Query Q15 pays for the unnecessary state keeping of the loop-lifted variant and Q16 exploits the new algorithm.

The second narrow bar from left in Figure 5.2 displays the performance improvement of the loop-lifted `descendant` path step (without loop-lifting the `child` axis). Queries Q1, Q4, Q15, and Q16 either contain no `descendant` step or evaluate such a location step on text nodes or attributes only. The `descendant` steps for most other queries are implicitly added by the translation of the `element` constructor. The only explicit `descendant` location steps are used in the queries Q6, Q7, and Q14. For queries Q6 and Q7 this speeds up the evaluation by more than a factor 2. In Q14 the full-text search mentioned earlier and the `child` path steps narrow the improvement.

The combination of both loop-lifted `child` and loop-lifted `descendant` step, illustrated in the second narrow bar from right, nicely adds up the performance improvements of the previous two versions. Ignoring queries Q1 and Q15 that suffer from the additional state keeping of the loop-lifted versions we obtained a performance gain by a factor 10 in most queries.

The early name and kind test optimization introduced in Section 4.2.4 is represented by the right-most narrow bar in Figure 5.2. Its application is most obvious in the queries Q1 and Q15. In both cases, we see a clear performance decrease. The reasons are the candidate lists, which are not selective enough. Aligning these lists becomes more expensive than

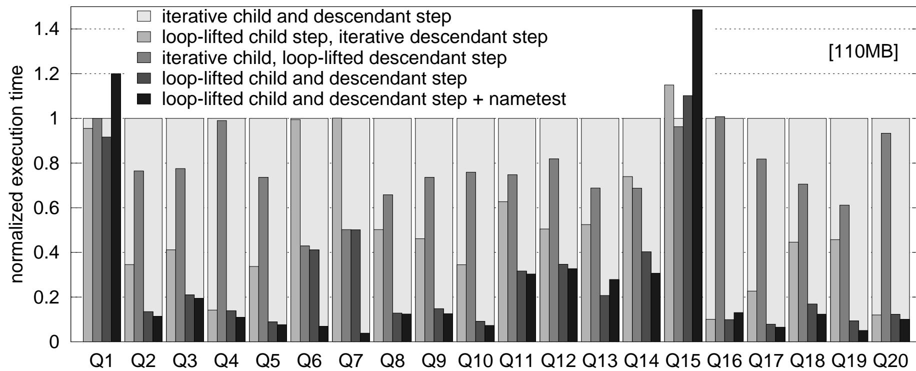


Figure 5.2: Benefits of loop-lifted staircase join. The four optimizations (narrow bars) show the performance gain in comparison to the iterative application of the basic staircase join. The loop-lifting of `child` and `descendant` step significantly speeds up the evaluation of most queries. The benefits of the early name tests can be seen by comparing the two right-most narrow bars.

applying a post-filter on the small results. On the other hand there are the queries Q6 and Q7 whose explicit `descendant` steps clearly benefit from the early name tests. Without the name test, these path steps produce quite large intermediate results that dominate the overall cost. Since the name test is quite selective for them, using early tests yields another factor 6 and 12 for Q6 and Q7, respectively, thus resulting in an overall improvement of factor 15 and 25, respectively. For all other queries the selection pushdown in the `child` location steps results in minor performance changes.

5.3 Interfaces

In comparison to the other optimizations, using the different interfaces does not introduce better evaluation strategies. Instead, it exploits more knowledge about the applied strategies and thus avoids unnecessary operations. In case of the value interface, the insertion with duplicate elimination can be avoided. For the node interface it is the additional path step as well as the storage overhead and for the path step interface the two order operations are not needed.

Therefore, no further justification for the application of these interfaces is necessary. Nevertheless we compared the evaluation times of query Q14 using the normal and the value interface for the input arguments of the `contains` function to get an idea what impact this optimization has. The value interface avoids the intermediate storage of an especially large part of the document in the string relation. The additional MIL code of the normal interface for the insertion, duplicate elimination, and value lookup required about 25 percent of the overall evaluation time for all document sizes (see Figure 5.3).

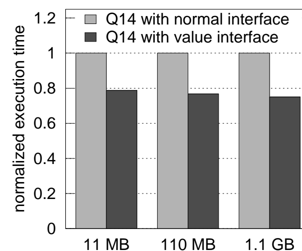


Figure 5.3: Benefits of avoiding normal interface. Queries Q14 performs about 25% faster for all sizes using the value interface.

5.4 Join Recognition

Without join detection, our implementation requires minutes to evaluate the XMark join queries (Q8–Q12) on the 11 MB XML document and larger instances do not finish within

hours, due to excessive resource consumption. The reason for this behavior is the generation of huge intermediate results in the size of Cartesian products, a consequence of loop-lifting. With the join optimization described in Section 4.4, we are able to significantly increase the performance. With join detection, XQuery join queries on large documents (11 GB) can be evaluated in interactive time. Figure 5.4 (using logarithmic scale) shows the results of the five XMark join queries with and without join recognition applied. For the queries Q8, Q9, and Q11 the performance improves by more than two orders of magnitude!

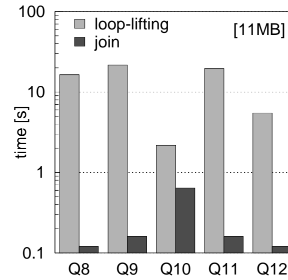


Figure 5.4: Benefits of XQuery join optimization. Queries Q8, Q9, and Q11 perform over a 100 times faster with join recognition applied.

5.5 Scaling

With the optimizations described in Chapter 4, we were not only able to improve our performance by up to three orders of magnitude, but also reached our goal of a scalable system. MonetDB/XQuery is able to query documents up to 11 GB size achieving linear performance scaling. Figure 5.5 shows the performance results normalized to the elapsed time on the 110 MB document (again with logarithmic scale). With the document sizes growing exponentially (11 MB, 110 MB, 1 GB, 11 GB), the graph shows that MonetDB/XQuery scales linearly with the document size. To be more precise, it scales with the size of the biggest intermediate result, which is limited in most cases by the document size.

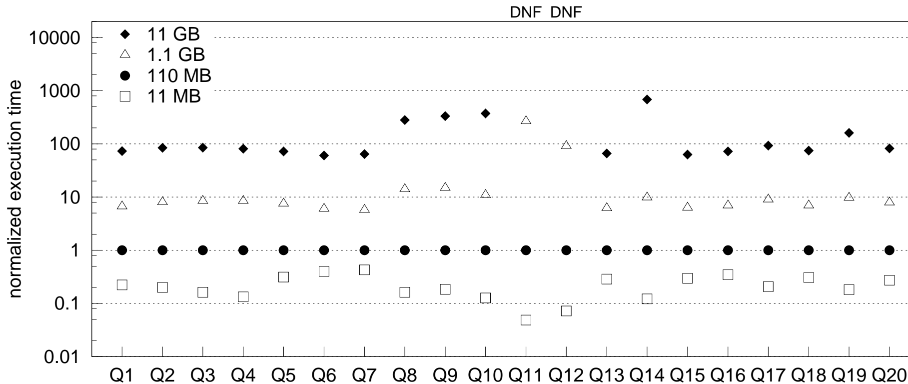


Figure 5.5: Scalability with respect to document size. All evaluation times are normalized to the time on the 110 MB document. The only quadratically scaling queries are Q11 and Q12, which did not finish the evaluation of the 11 GB document.

While most queries scale perfectly for the sizes above 110 MB, they seem to gain less than factor 10 from 11 to 110 MB. The reasons are the parsing costs of the large MIL scripts as well as the initialization costs. These fixed costs slow down the otherwise very fast queries (*e.g.*, query Q6 evaluates the 110 MB instance in just 50 ms). For larger document sizes these costs become negligible.

The queries, which do not scale perfectly linear with the document size, are the join queries (Q8–Q12), whose join result is bigger than the input document, and query Q14, where excessive memory consumption makes our system start swapping to hard disk for the 11 GB document. Two of the five join queries (Q11 and Q12) even fail to complete on the 11 GB document size. The bottleneck in both queries is a theta-join (comparison via $>$) that generates an intermediate result with about 120G tuples for the 11 GB document

(with 16G tuples). Thus *any* XQuery system must necessarily exhibit quadratic scaling with document size on the query Q11 and Q12.

5.6 System Comparison

After showing the advantages of the applied optimizations and the resulting scalability, we will now compare our approach against two other XQuery systems. The test set is once more the XMark benchmark at scaling factors 0.1 to 100 (which yield documents of respectively 11 MB to 11 GB).

The first reference system is the latest version of Galax (0.5.0) as the most popular and well known ”native” XQuery engine in open-source [11]. Galax is a file-oriented implementation, which parses the referenced XML documents for each query. This often dominates the evaluation time. To compare fairly, we used the monitor feature of Galax and selected only the running times for the evaluation phase.

The second reference system, a native XML database system, is the current version of X-Hive (X-Hive/DB 6.0), which claims to be the ”The fastest and most scalable XML database powered by open standards” [26]. Performance figures in [9] attest X-Hive a high scalability, which convinced us to include it in our evaluation. For X-Hive again we measured only the query evaluation time.

		11 MB			110 MB			1.1 GB		11 GB
operation:	Q	Galax	X-Hive	Pf/M	Galax	X-Hive	Pf/M	X-Hive	Pf/M	Pf/M
selection	1	0.06	0.37	0.04	0.72	1.29	0.18	9.9	1.2	13
positional predicate	2	0.03	0.45	0.06	0.31	1.75	0.30	33.0	2.4	25
value comparison	3	0.14	0.65	0.24	1.76	5.66	1.48	25.1	12.5	126
document order	4	0.22	0.10	0.06	2.91	1.00	0.45	18.1	3.8	36
implicit casting	5	0.05	0.13	0.05	0.63	0.90	0.16	20.7	1.2	11
descendant steps	6	1.30	1.07	0.02	13.29	10.17	0.05	178.1	0.3	3
	7	2.68	1.57	0.03	30.01	24.84	0.07	278.4	0.4	4
join	8	0.16	0.85	0.12	2.12	3.51	0.74	49.1	10.4	208
nested joins	9	113.23	32.25	0.16	*DNF*	12280.66	0.87	*DNF*	12.9	289
join, element constructions	10	1.74	5.28	0.64	18.61	442.37	5.05	*DNF*	55.0	1882
joins on generated values	11	2.62	98.91	0.16	*DNF*	19927.29	3.28	*DNF*	872.5	*DNF*
	12	1.44	23.39	0.12	*DNF*	5100.19	1.66	*DNF*	150.7	*DNF*
result reconstruction	13	0.03	0.10	0.06	0.66	1.03	0.21	12.9	1.3	13
full text search	14	1.92	0.72	0.17	99.53	11.16	1.40	110.2	13.7	959
long path (child steps), fn:not, fn:empty	15	0.02	0.03	0.08	0.20	0.49	0.27	10.6	1.7	16
	16	0.03	0.03	0.09	0.46	0.52	0.26	10.9	1.8	18
	17	0.06	0.09	0.06	0.82	0.85	0.29	11.8	2.6	26
user-defi ned functions	18	0.07	0.08	0.04	0.73	0.64	0.13	14.8	0.9	9
order by	19	1.17	0.67	0.10	14.73	12.15	0.55	254.5	5.3	88
aggregation	20	0.28	0.11	0.17	2.98	1.40	0.62	24.6	4.9	50

Table 5.1: Overview of XMark query evaluation times (elapsed time in seconds).

Table 5.1 lists our full experimental results. Galax numbers are available only for the 11 MB and 110 MB XML documents, since it fails to load the 1 GB file. For the simpler queries on the 11 MB file, Galax is on par with the other systems and sometimes performs fastest although by a small margin. For the same queries on the 110 MB document Galax already looses some ground, but still completes the evaluation in reasonable time. The more expensive queries for Galax are the ones with `descendant` steps (Q6 and Q7), the join queries (Q8–Q12), the full text search (Q14), and the `order by` implementation (Q19). While Galax seems to spot the joins in queries Q8 and Q10, it fails to do so otherwise. Again the performance difference on these two queries increases with growing document size. The other three join queries crashed on the 110 MB document with *materialization out of bounds* errors, most probably due to the quadratic join complexity.

X-Hive behaves similar to Galax for most queries. We could speed up the evaluation on a number of queries by creating value indices on the paths `buyer/@person` and

`profile/@income`. Thanks to these indices it evaluates the join query Q8 fast as well. However, the quadratic performance on Q9–Q12 indicates that such indices only help on a small class of queries. As soon as queries join intermediate query results, indices cannot be used and performance degrades strongly.

For the queries with descendant steps (Q6, Q7, and Q14) our implementation of loop-lifted staircase join clearly outperforms both other systems. The same applies for the join queries (Q8–Q12), where we can benefit from our join recognition logic. The remaining queries, whose main tasks are database operations (full text search in Q14 and `order by` in query Q19), furthermore show the big difference between the mature algorithms of the relational database MonetDB, the stand-alone implementation Galax and the native database X-Hive.

Chapter 6

Conclusions and Outlook

This work builds on an XPath aware relational encoding of XML trees [13, 17] and a relational XQuery compiler [15, 16] to turn a relational database back-end into an XQuery processor. It combines the concepts of the XML document encoding and the XQuery sequence encoding in a storage scheme, which is applicable in the MonetDB database. Based on this relational encoding, we enhanced the inference rules described in [16] with more explicit storage information. Furthermore we extended the mapping with additional XQuery constructs (*e.g.*, conditionals, constructors, and `order by`) to support large parts of the XQuery language. The mapping of XQuery to MIL in Section 3.5, which also corresponds to the practical part of this thesis, applied all these ideas. Together with the optimizations developed in this work, we delivered not only a proof of concept but also created a fast and highly scalable XQuery system [3].

Other attempts of building an XQuery processor on top of a relational database like *e.g.*, Microsoft's *SQL Server 2005* [24] extend the relational back-end with additional operators that cope with the different XQuery constructs (*e.g.*, the `for` expression). Such implementations as well as native XML database systems could overcome these specific XQuery operators by integrating our compilation scheme in their work. The changes would be restricted to the document storage, the path step evaluation and the node constructors. The support of XQuery and SQL/XML [18] would then be a matter of additional optimization rules instead of a new, full-fledged evaluation engine.

We believe, however, that exchanging the encoding and storage of XML documents as well as the relational back-end MonetDB may provide a significant performance decrease. The reason is that the dense preorder ranks of our XML document encoding support the XPath location steps especially well. In MonetDB, evaluating path steps maps to positional lookups in arrays. If our system was backed just by ordinary B-tree index lookups, we expected a significant drop in performance. In contrast, variable-length surrogates such as, ORDPATH labels [23], are designed to allow "low-cost" updates while still encoding document order. However, fast updates come at the expense of higher storage and manipulation costs (positional skipping is not possible and index lookups must be used instead).

In our pre/size encoding, structural updates on the other hand are considered to be problematic (*i.e.*, physical cost linear to document size), because they cause shifts in all preorder ranks after the update point. Furthermore, they require updates of the size values of all ancestors, such that the root of the tree becomes a locking bottleneck. In [4], we showed how such problems can be avoided by updating the size of ancestors using delta-increments, which are transaction-commutative operations. The proposal additionally reduces the physical cost to the minimum (*i.e.*, linear to update volume) by carefully exploiting the virtual column feature of MonetDB to store pre numbers.

In Chapter 4, we focused on the optimizations that promised to improve the implementation most. In [9], where the authors describe an XQuery compiler that was originally designed to emit SQL code, two of these optimizations, namely built-in order-awareness and

XQuery join recognition were also recognized as key features to process XML documents of serious size. While their XQuery join pattern resembles the query pattern discussed in Section 4.4, it largely remains unclear how to derive a relational join plan. Performance-wise, we really reap the benefits of using an extensible RDBMS kernel as an XQuery runtime environment: the XMark benchmark figures obtained here surpass those reported in [9] by two orders of magnitude.

While these optimizations improve our implementation significantly, a great number of optimizations has not yet been explored. One decision that we never compared with other opportunities was the design of our storage scheme. It is therefore not clear whether splitting up all document fragments into different containers or one big set of relations would be a better choice. We also tested only scenarios with single documents. It could be interesting to analyze queries, which operate on a large collection of documents.

In this work, the elimination of common sub-expressions is done in a very naive way, which leaves a large number of them undetected. In Pathfinder/MonetDB, variable bindings of XQuery variables are evaluated once. This, however, restricts the number of eliminated common sub-expressions only to variables in the XQuery expression. In XMark query Q20, for example, the same path steps are evaluated three times. An equivalent version that binds these steps to a variable would perform faster. Other common sub-expression eliminations occur within XQuery constructs, where the same expressions (*e.g.*, the input sequence of a `for` loop) are evaluated only once, and between XQuery constructs, where the expressions of the different loop relations are evaluated once. A real common sub-expression elimination thus would certainly improve our implementation. The impact, however, is yet unknown.

With the decision to keep all intermediate results sorted on iter and pos, we restrict ourself again to a naive mapping. The keyword `unordered`, aggregates and XPath location steps for example could utilize more algorithms without our order constraints. More work on this topic will be required in the future.

The join recognition is another optimization, which is far from complete. The main problem is that the join pattern works on normalized XQuery core expressions. While our patterns match a large number of XQuery joins, they will be always inferior to simplified patterns on relational algebra. In [14], the author sketches the properties necessary for such more robust join patterns.

While this work already provides a fast and scalable XQuery processor, the implementation does not offer enough support for a large number of optimizations: With the mapping to MIL, we loose too much context information, whereas normalized XQuery Core may be too abstract for some optimizations. As an intermediate step between the normalized XQuery Core and MIL, we will therefore use an annotated relational algebra, like the one described in Section 3.2, in the future. As the mapping rules from XQuery Core to relational algebra are already provided in Section 3.4, we can focus on the optimization of the relational algebra.

Algebraic optimizations have proven to be very successful in relational database management systems and various approaches to implement XQuery processors also rely on algebras as the basis for possible efficient optimizations. Timber [19], Natix [12], and Galax [11] *e.g.*, all use algebras that are specific to their respective XQuery system. These algebras try to utilize the special features of XQuery.

In contrary, our approach — using pure relational algebra — leverages mature optimization techniques of relational systems. Properties like (multi-valued) dependencies and constant relations, which were already used in the context of relational systems, enable efficient optimizations (*e.g.*, join recognition) in our approach as well [14]. Additionally other XQuery specific annotations like type and schema information might become useful. Moreover valuable information from a more abstract level, that is required for optimizations when finally generating the physical query plans, may be stored as annotation.

The optimized relational algebra expression easily maps to an underlying database language. While this may be any database that understands relational algebra plans, we will

stick to MonetDB, which in the current version of our XQuery processor already proves to be a good choice. Until this effort is completed, MonetDB/XQuery, the current implementation of Pathfinder/MonetDB described in this thesis, will provide a competitive and scalable relational runtime for XQuery.

Bibliography

[1] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. World Wide Web Consortium, Apr. 2005. `http://www.w3.org/TR/xquery-use-cases/`.

[2] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, UVA, Amsterdam, The Netherlands, May 2002.

[3] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: XQuery—The Relational Way. In *Proc. VLDB Conf.*, 2005. Demo.

[4] P. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proc. XIME-P*, 2005.

[5] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. World Wide Web Consortium, Apr. 2005. `http://www.w3.org/TR/xquery/`.

[6] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. Lecture Notes in Computer Science. Springer-Verlag, Dec. 2000.

[7] A. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), June 1970.

[8] G. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD Conf.*, pages 268–279, Austin, TX, USA, May 1985.

[9] D. DeHaan, D. Toman, M. Consens, and M. Öszu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding. In *Proc. SIGMOD Conf.*, pages 623–634, San Diego, CA, USA, June 2003.

[10] E. Elmasri and A. Navathe. *Fundamentals of Database Sytems*. Benjamin/Cummings, Redwood City, CA, USA, 2nd edition, 1994.

[11] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proc. VLDB Conf.*, pages 1077–1080, Berlin, Germany, Sept. 2003.

[12] T. Fiebig and G. Moerkotte. Algebraic xml construction in natix. In *WISE '01: Proceedings of the Second International Conference on Web Information Systems Engineering (WISE'01) Volume 1*, page 212, Washington, DC, USA, 2001. IEEE Computer Society.

[13] T. Grust. Accelerating XPath Location Steps. In *Proc. SIGMOD Conf.*, pages 109–120, Madison, WI, USA, June 2002.

[14] T. Grust. Purely Relational FLWORs. In *Proc. XIME-P*, Baltimore, Maryland, June 2005.

[15] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB Conf.*, pages 252–263, Toronto, Canada, Sept. 2004.

[16] T. Grust and J. Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Twente Data Management Workshop on XML Databases and Information Retrieval (TDM)*, pages 7–14, Enschede, The Netherlands, June 2004.

[17] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. VLDB Conf.*, pages 524–535, Berlin, Germany, Sept. 2003.

[18] International Organization for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML).

[19] H. Jagadish, L. Lakshamnan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. DBPL*, pages 149–164, Frascati, Italy, Sept. 2001.

[20] H. Korth and A. Silberschatz. *Database Systems Concepts*. McGraw-Hill, Inc., New York, San Francisco, Washington, DC, USA, 1991.

[21] MonetDB. `http://monetdb.cwi.nl/`.

[22] MonetDB/XQuery. `http://monetdb-xquery.org/`.

[23] P. O'Neil, O. E.J., S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATH: Insert-Friendly XML Node Labels. In *Proc. SIGMOD Conf.*, pages 903–908, Paris, France, June 2004.

[24] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, P. Kukol, W. Yu, D. Tomic, A. Baras, C. Kowalczyk, B. Berg, D. Churin, and E. Kogan. XQuery Implementation in a Relational Database System. In *Proc. VLDB Conf.*, Trondheim, Norway, Aug. 2005.

[25] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB Conf.*, pages 974–985, Hong Kong, China, Aug. 2002.

[26] X-Hive/DB. `http://www.x-hive.com/`.

Acknowledgments

This work was only possible with the help of many people, who supported me in various ways. I take this opportunity to express my thanks to all of them.

In the summer semester 2002, I first learned about XML, XPath, and XQuery. It was in the course *XML* that was lectured by Torsten Grust. With his entertaining and competent manner, he did not only arouse my interest in XML, but also caught my attention for his lectures in the following semesters. I learned a lot from him and eventually joined his research group to write my bachelor thesis. It was also Torsten, who arranged my internship at the CWI in Amsterdam and mentored me during all that time.

During my internship at the CWI in April 2004 I started with the implementation of the work described in this thesis. Peter Boncz, who supervised me during my stay, was always available for insightful discussions. With his sometimes unconventional but in most cases ingenious ideas, he always found a way whenever we seemed to have reached a dead end. With Stefan Manegold I shared many long evenings at the CWI. We discussed Pathfinder and MonetDB as well as everything else for many hours. Martin Kersten, who — as team leader — made this internship possible, always encouraged me in my work even if I sometimes lost my faith. Together with all others from the CWI they made this internship an unforgettable experience, which I enjoyed much.

Back in Konstanz the Graduiertenkolleg and especially Marc Scholl supported me with excellent facilities to present our research and to continue my work. During that time Jens Teubner's door was the one I most frequently knocked at. He already helped me during my bachelor studies when I visited Torsten's courses. During all the time I learned a lot from him. He was always available if I needed any advice, comments, or help (even if it was not related to the current work). He is also the one who made the thesis in this form possible. His numerous comments and suggestions for improvement were a tremendous support.

I want to thank all of them as well as all the others I left unmentioned for everything they have done for me. I enjoyed working in such inspiring environments and hope for many more years of exciting joint research.

Furthermore, I want to thank my friends, who accompanied me during the last years, and my family. I had a lot of fun and without them I would not have had such a good time. Above all, thanks to Sylvie, who endured long periods of separation, always supported me in all decisions and lovingly accepted many working hours :-).

Appendix A

XMark Queries

XMark Q1

```
for $b in doc("auction.xml")/site/people/person[@id = "person0"]
return $b/name/text()
```

XMark Q2

```
for $b in doc("auction.xml")/site/open_auctions/open_auction
return
  <increase>
    { $b/bidder[1]/increase/text() }
  </increase>
```

XMark Q3

```
for $b in doc("auction.xml")/site/open_auctions/open_auction
where zero-or-one($b/bidder[1]/increase/text()) * 2
               <= $b/bidder[last()]/increase/text()
return
  <increase
    first="{ $b/bidder[1]/increase/text() }"
    last="{ $b/bidder[last()]/increase/text() }"
  />
```

XMark Q4

```
for $b in doc("auction.xml")/site/open_auctions/open_auction
where
  some $pr1 in $b/bidder/personref[@person = "person20"],
       $pr2 in $b/bidder/personref[@person = "person51"]
  satisfies $pr1 << $pr2
return
  <history>
    { $b/reserve/text() }
  </history>
```

XMark Q5

```
count(
  for $i in doc("auction.xml")/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

XMark Q6

```
for $b in doc("auction.xml")//site/regions
return
  count($b//item)
```

XMark Q7

```
for $p in doc("auction.xml")/site
return
  count($p//description) + count($p//annotation) + count($p//emailaddress)
```

XMark Q8

```
for $p in doc("auction.xml")/site/people/person
let $a :=
  for $t in doc("auction.xml")/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return
  <item person="{$p/name/text()}">
    { count($a) }
  </item>
```

XMark Q9

```
for $p in doc("auction.xml")/site/people/person
return
  <person name="{$p/name/text()}">
    {
      for $t in doc("auction.xml")/site/closed_auctions/closed_auction
      where $p/@id = $t/buyer/@person
      return
        <item>
          {
            let $n :=
              for $t2 in doc("auction.xml")/site/regions/europe/item
              where $t/itemref/@item = $t2/@id
              return $t2
            return $n/name/text()
          }
        </item>
    }
  </person>
```

XMark Q10

```
for $i in distinct-values(doc("auction.xml")/site/people/
                          person/profile/interest/@category)
return
  <categorie>
    {
      <id>{$i}</id>
      ,
      for $t in doc("auction.xml")/site/people/person
      where $t/profile/interest/@category = $i
      return
```

```
        <personne>
          <statistiques>
            <sexe>{$t/profile/gender/text()}</sexe>
            <age>{$t/profile/age/text()}</age>
            <education>{$t/profile/education/text()}</education>
            <revenu>{fn:data($t/profile/@income)}</revenu>
          </statistiques>
          <coordonnees>
            <nom>{$t/name/text()}</nom>
            <rue>{$t/address/street/text()}</rue>
            <ville>{$t/address/city/text()}</ville>
            <pays>{$t/address/country/text()}</pays>
            <reseau>
              <courrier>{$t/emailaddress/text()}</courrier>
              <pagePerso>{$t/homepage/text()}</pagePerso>
            </reseau>
          </coordonnees>
          <cartePaiement>{$t/creditcard/text()}</cartePaiement>
        </personne>
    }
  </categorie>
```

XMark Q11

```
for $p in doc("auction.xml")/site/people/person
let $l :=
  for $i in doc("auction.xml")/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
return
  <items name="{$p/name/text()}">
    { count($l) }
  </items>
```

XMark Q12

```
for $p in doc("auction.xml")/site/people/person
let $l :=
  for $i in doc("auction.xml")/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
where $p/profile/@income > 50000
return
  <items person="{$p/profile/@income}">
    { count($l) }
  </items>
```

XMark Q13

```
for $i in doc("auction.xml")/site/regions/australia/item
return
  <item name="{$i/name/text()}">
   { $i/description }
  </item>
```

XMark Q14

```
for $i in doc("auction.xml")/site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()
```

XMark Q15

```
for $a in
  doc("auction.xml")/site/closed_auctions/closed_auction/annotation/
  description/parlist/listitem/parlist/listitem/text/emph/keyword/text()
return
  <text>
    { $a }
  </text>
```

XMark Q16

```
for $a in doc("auction.xml")/site/closed_auctions/closed_auction
where
  not(
    empty(
      $a/annotation/description/parlist/listitem/
      parlist/listitem/text/emph/keyword/text()
    )
  )
return
  <person id="{$a/seller/@person}"/>
```

XMark Q17

```
for $p in doc("auction.xml")/site/people/person
where empty($p/homepage/text())
return
  <person name="{$p/name/text()}"/>
```

XMark Q18

```
declare namespace local = "http://www.example.org";
declare function local:convert($v as xs:decimal?) as xs:decimal?
{
  2.20371 * $v (: convert Dfl to Euro :)
};

for $i in doc("auction.xml")/site/open_auctions/open_auction
return
  local:convert(zero-or-one($i/reserve))
```

XMark Q19

```
for $b in doc("auction.xml")/site/regions//item
let $k := $b/name/text()
order by $b/location ascending empty greatest
return
  <item name="{$k}">
   { $b/location/text() }
 </item>
```

XMark Q20

```
<result>
  <preferred>
    {
      count(
        doc("auction.xml")/site/people/person/
        profile[@income >= 100000]
      )
    }
  </preferred>
  <standard>
    {
      count(
        doc("auction.xml")/site/people/person/
        profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
  <challenge>
    {
      count(
        doc("auction.xml")/site/people/person/
        profile[@income < 30000]
      )
    }
  </challenge>
  <na>
    {
      count(
        for $p in doc("auction.xml")/site/people/person
        where empty($p/profile/@income)
        return $p
      )
    }
  </na>
</result>
```