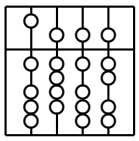
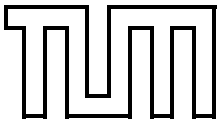


Technische Universität München
Institut für Informatik
Lehrstuhl für Datenbanksysteme



Pathfinder: XQuery Compilation Techniques for Relational Database Targets

Jens Thilo Teubner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Helmut Seidl

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Torsten Grust
2. Prof. Dr. Martin L. Kersten,
   Universiteit van Amsterdam/Niederlande

Die Dissertation wurde am 10. Mai 2006 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28. September 2006 angenommen.

This thesis is also available in print (ISBN 3-89963-440-3).

pathfinder (ˈ**pɑːθˌfaɪndə**) *n.* a person who makes or finds a way, esp. through unexplored areas or fields of knowledge.

Collins English Dictionary [Makins95]

Abstract

Even after at least a decade of work on XML and semi-structured information, such data is still predominantly processed in main-memory, which obviously leads to significant constraints for growing XML document sizes.

On the other hand, mature database technologies are readily available to handle vast amounts of data easily. The most efficient ones, relational database management systems (RDBMSs), however, are largely locked-in to the processing of very regular, table-shaped data only. In this thesis, we will unleash the power of *relational* database technology to the domain of *semi-structured* data, the world of XML. We will present a purely relational XQuery processor that handles huge amounts of XML data in an efficient and scalable manner.

Our setup is based on a relational *tree encoding*. The *XPath accelerator*, also known as *pre/post* numbering, has since become a widely accepted means to efficiently store XML data in a relational system. Yet, it has turned out that there is additional performance to gain. A close look into the encoding itself and the deliberate choice of relational indexes will give intriguing insights into the efficient access to node-based tree encodings.

The backbone of XML query processing, the access to XML document regions in terms of XPath tree navigation, becomes particularly efficient if the database system is equipped with enhanced, *tree-aware* algorithms. We will devise *staircase join*, a novel join operator that encapsulates knowledge on the underlying tree encoding to provide efficient support for XPath navigation primitives. The required changes to the DBMS kernel remain remarkably small: staircase join integrates well with existing features of relational systems and comes at the cost of adding a single join operator only. The impact on performance, however, is significant: we observed speedups of several orders of magnitude on large-scale XML instances.

Although existing XQuery processors make use of the resulting XPath performance gain by evaluating XPath navigation steps in the DBMS back-end, they tend to perform other core XQuery operations outside the relational database kernel. Most notably this affects the `FLWOR` *iteration* primitive, XQuery's *node construction* facilities, and the *dynamic type* semantics of XQuery. The *loop-lifting* technique we present in this work deals with these aspects in a purely relational fashion. The outcome is a *loop-lifting compiler* that translates *arbitrary* XQuery

expressions into a *single* relational query plan. Generated plans take particular advantage of the operations that relational databases know how to perform best: relational *joins* as well as the computation of *aggregates*.

The *MonetDB/XQuery* system to which this thesis has contributed provides the experimental proof of the effectiveness of our approach. MonetDB/XQuery is one of the fastest and most scalable XQuery engines available today and handles queries in the multi-gigabyte range in interactive time. The key to this performance are the techniques described in this thesis. They form the basis for MonetDB/XQuery's core component: the XQuery compiler *Pathfinder*.

Contents

1 Introduction

Since 1998, when the W3 Consortium published its first XML Recommendation, the Extensible Markup Language has become a standard means for data representation, storage, and interchange. The XML format has proven to be versatile enough to describe virtually any kind of information, ranging from a couple of bytes in Web Service messages to gigabyte-sized data collections (*e.g.*, [Ley, PIR]).

The massive amount of data available in the XML format raises an increasing demand to store, process, and query these data in an effective manner. The W3 Consortium has long since realized this demand and since 1999, the XML Query Working Group has been developing a standard query language for XML: *XQuery* [Boag05]. Its proposals have matured over the past years and the first official W3C XQuery Recommendation is expected to be released soon.

XQuery is a functional and strongly typed language built around its (older) sublanguage *XPath*. Other core functionalities include the `FLWOR` (pronounced "flower") looping primitive, conditionals, and a means to construct transient XML nodes during query processing, turning XQuery into a Turing-complete language [Kepser04].

Current XML processors mainly implement XPath, XSLT, and XQuery processing in main memory. Increasing amounts of XML data, however, render this approach infeasible. Data manipulation (updates), transaction management, as well as security issues raise additional questions. All these concerns have quite early suggested the use of *database technology* to process XML data.

1.1 Database Technology for XML

Database systems, *e.g.* for relational data, employ techniques that provide scalability into the terabyte data range and beyond. Applied to the XML domain, these techniques promise a similar scalability for XML query processing.

There are a number of key aspects that lead to the scalability of modern database management systems:

(i) Suitable storage model.
Data volumes that exceed the limits of main memory require appropriate storage structures that easily extend to secondary storage devices (*e.g.*, hard drives). An example is the avoidance of physical addresses for data references in favor of logical identifiers.

(ii) Suitable algorithms.
Database algorithms are designed for operation on secondary storage. The adherence to specific data *access patterns* allows for intelligent caching and storage management.

(iii) Declarative query formulation.
The use of a declarative (intermediate) query language decouples a user's request from specific physical implementations. This allows for the interchangeability of physical operators, the construction of access and index structures, and the rewriting of query plans without affecting the user interface or the query outcome. The latter is facilitated by an *algebraic* intermediate query representation with mathematically sound rewrite rules.

Experiences on relational database systems show that it is particularly beneficial to express queries in a *bulk-oriented* (rather than tuple-oriented) fashion. This holistic view allows operators to be implemented for highly cache-friendly behavior.

(iv) Query optimization.
The choice and execution order of database operators can have a significant impact on query performance. Database systems, hence, employ sophisticated query optimizers that rewrite queries according to rules, heuristics, and physical cost models. An appropriate intermediate query representation facilitates optimization, too.

1.1.1 Native XML Databases

The approaches to bring database technology into the XML domain turn out to be quite diverse. An obvious approach is to build up a database system from scratch

that employs XML trees as its intrinsic data type. Such XML database systems are usually referred to as *native* XML databases.

A representative of a native XML database is the Natix system, developed by Fiebig *et al.* [Fiebig02]. Natix implements a novel storage engine to account for the XML tree structure. In a nutshell, documents are semantically split based on their tree structure. Tree fragments are then stored on disk pages of fixed size. The split algorithm may be tuned using a *split matrix* to accommodate for specific application needs. Full text and the structural XASR[1] indexes may be created to further accelerate tree access.

Natix offers a bulk-oriented intermediate query representation in terms of the Natix Physical Algebra (NPA), which operates on sequences of tuples. XML tree operations as well as XQuery's `FLWOR` iteration primitive are handled as nesting and unnesting operations in NPA.

The Timber system by Jagadish *et al.* [Jagadish02] takes a slightly different approach to an algebraic query description. TAX, a tree algebra for XML [Jagadish01], defines manipulations on multi-sets of trees. To accommodate for the heterogeneity in XML trees, TAX introduces the notion of *pattern trees*. As an annotation to each tree set, a pattern tree describes the commonality of all the trees in the set. The common features may then be used to efficiently represent all items as homogeneous tuples.

TAX pushes declarative query description even a step further. The access to XML tree nodes is described in terms of *pattern matching* of a pattern tree against an XML tree, which might be appropriately supported by, *e.g.*, structural join algorithms [Bruno02]. A concise mapping from arbitrary XPath expressions to respective pattern trees has never been formally published, though.

Only recently, IBM has presented the Viper system [Nicola05] that will form the XML back-end for upcoming releases of the DB2 Universal Database® system. Similar to Natix, Viper splits up the XML document tree and distributes tree fragments over database pages of fixed size.

1.1.2 Relational Back-Ends

Building up a completely new database engine for XML from scratch almost seems like a re-invention of the wheel. The past decades of research have turned relational databases into highly efficient data processors that readily provide effective optimizers, fault tolerant storage techniques, and concurrency control. The relational data model has proven its versatility to process virtually any kind of information.

With this insight, an avalanche of papers has been published that describe relational storage techniques for XML data. In their survey paper, Krishnamurthy

[1] Extended Access Support Relation

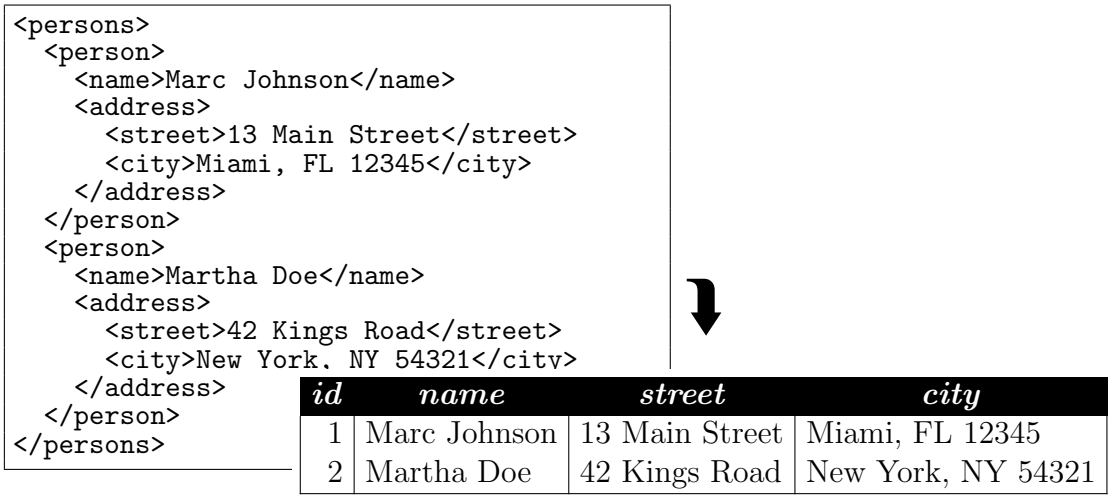


Figure 1.1: Schema-based mapping of XML documents to relational tables.

et al. [Krishnamurthy03] classify these proposals into *schema-based* and *schema-oblivious* mapping schemes.

Schema-Based Storage

The idea of schema-based XML storage is illustrated in Figure 1.1: an input XML document is mapped to a relational table whose schema reflects the semantic content of the document. The schema is typically derived from a DTD or XML Schema specifications (*e.g.,* [Shanmugasundaram99]), or from the specific data instance [Deutsch99]. These schema-based approaches are closely related to *XML publishing*, where relational data is serialized into an XML format and then queried using, *e.g.*, XQuery. The bidirectional mapping between XML and the relational model becomes particularly important in the context of *information integration* and eases the creation of value-based indexes.

However, intrinsic shortcomings of schema-based XML storage approaches rule out their application for a versatile and standards-compliant XQuery engine. *Document order*, an inherent concept in the XML data model, is only recoverable if multiple attributes are added to the resulting table schema. XPath navigation along *recursive axes* or with *wildcard name tests* lead to expensive multi-way unions and recursion, if possible at all. The implementation of XQuery expressions beyond XPath (`FLWOR` clauses, sequence order, or transient node construction) raises further questions. Finally, this approach depends on a strong regularity and schema conformance of the XML documents to store.

Schema-Oblivious Storage

An alternative approach is the schema-oblivious XML storage on relational back-ends. This approach stores the semantic properties of each XML tree node in a relational table of fixed schema, independently of any XML schema information. One or more attributes in the schema are reserved for the *structural* information of the XML tree.

A large number of proposals have been published to store XML documents in a schema-oblivious fashion. The ones that provide acceptable performance typically use *numbering schemes* to encode the XML tree structure and can be classified into one of two groups:

(i) *Dewey-based* schemes assign to each node a vector that represents its path from the document root (*e.g.*, ORDPATH [O'Neil04]), whereas

(ii) *pre/post-based* numberings use each node's ranks within a pre- and post-order tree walk to encode the structural information (*e.g.*, XPath accelerator [Grust02]).

We will elaborate on both approaches in the course of this thesis.

It turns out that the simplicity of the relational data model and the effectiveness of modern database technology easily compensate for the seemingly large mapping overhead and turn relational databases into highly efficient XML processors.

Relational XQuery

Schema-oblivious mapping techniques thus provide promising storage solutions as we demanded for database-backed XQuery evaluation in our list of key aspects on page 2. Existing work, however, cannot provide convincing solutions for the remaining three items in our list:

(ii) *Suitable algorithms.*
Conventional relational database kernels cannot fully exploit information on the XML tree structure encoded in relational tables and behave suboptimal if we demand close adherence to the XPath semantics.

(iii) *Declarative query formulation.*
Though it is not uncommon to keep XML data in a relational back-end, existing XQuery processors tend to perform core XQuery functionalities using imperative programming languages outside the database kernel. If we express incoming XQuery expressions in a relational algebra dialect as a whole, the entire query may be shipped to the back-end in one go and take full advantage of the RDBMS's bulk processing capabilities.

(iv) Query optimization.
The use of a relational back-end makes relational query optimization techniques available for XQuery processing. Nevertheless, we can expect further improvements if we consider specific properties of plans that originate from XQuery input.

1.2 Contributions of this Thesis

This thesis focuses on *relational* XQuery evaluation. We will derive a number of techniques that bridge the apparent gap between the relational model (based on *sets of tuples*) and the XQuery language with *sequences of items* as its underlying data model.

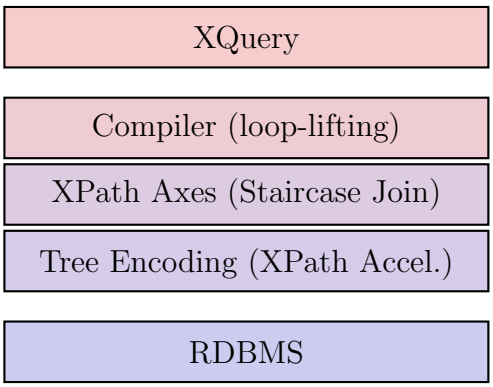


Figure 1.2: XQuery processing stack.

Together, these techniques assemble into the *purely relational XQuery processing stack*, shown on the left.

The thesis starts off with a refresher on the XPath accelerator numbering scheme [Grust02] in Chapter 2. The discussion will identify a number of interesting properties of this encoding that may aid the efficient query evaluation on existing relational back-ends. A thorough experimental section assesses how many of these properties off-the-shelf RDBMS implementations (IBM DB2 and PostgreSQL for that matter) can already grasp for efficient XPath processing. Parts of these results have already been published in a follow-up paper to the original XPath accelerator publication:

[Grust04e] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath Evaluation in any RDBMS. *ACM Transactions on Database Systems (TODS)*, 29(1), pages 91–131, March 2004.

In terms of their advanced index structures (namely B-tree or R-tree indexes), existing systems can benefit from the proposed tree encoding to a remarkable extent. However, the statistical information collected by conventional RDBMSs cannot capture specifics in the generated data distribution that stem from the fact that the relational tables actually encode a tree; operators are unable to exploit these specifics for optimized query processing.

This takes us to the development of a novel relational join operator, the *staircase join*, in Chapter 3. Staircase join encapsulates full tree awareness for XML

trees encoded in relational tables. The algorithm can be plugged into a relational database kernel just like any other join operator, requiring only local modifications to the kernel. Query rewrite techniques, such as selection pushdown, are still available with staircase join. The original algorithm has been published in:

[Grust03c] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, pages 524–535. Berlin, Germany, September 2003.

[Grust03b] Torsten Grust, Maurice van Keulen, and Jens Teubner. Bridging the Gap Between Relational and Native XML Storage with Staircase Join. In *Proc. of the 15th GI Workshop on Foundations of Database Systems*, pages 85–89. Tangermünde, Germany, June 2003.

To support the claim that staircase join can speed up XPath processing in *any* RDBMS, we incorporated the algorithm into the open source database PostgreSQL, as published in:

[Mayer04b] Sabine Mayer, Torsten Grust, Maurice van Keulen, and Jens Teubner. An Injection with Tree Awareness: Adding Staircase Join to PostgreSQL. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 1305–1308. Toronto, Canada, September 2004.

Only lately, we have extended staircase join to also handle the XPath evaluation of multiple context sequences in a single run. This *loop-lifted staircase join* forms the basis for a full-scale XQuery implementation:

[Boncz05b] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Loop-Lifted Staircase Join: From XPath to XQuery. Technical Report INS-E0510, CWI, Amsterdam, March 2005.

While the application of staircase join leads to a highly efficient XPath step evaluation, the remaining features of the XQuery language are often left to a standard programming language outside the DBMS kernel. Most notably in this respect are XQuery's `FLWOR` iteration primitive and transient node construction.

In Chapter 4, we will push relational XML processing one step further and extend our processing stack to full XQuery compliance. Our approach remains purely relational: our *loop-lifting* compilation procedure turns arbitrary XQuery expressions into purely relational plans, efficiently executable on, *e.g.*, SQL hosts. An overview of the compilation procedure has been published as:

[Grust04c] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 252–263. Toronto, Canada, September 2004.

[Grust04d] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Proc. of the 1st Twente Data Management Workshop (TDM)*, pages 7–14. Enschede, The Netherlands, June 2004.

To assess the viability of our approach in practice, the open-source implementation *Pathfinder* accompanies this thesis. Pathfinder implements the full processing stack and compiles XQuery expressions into code for the relational back-end MonetDB [Boncz02]. The compiler is part of the *MonetDB/XQuery* system, one of the fastest and most scalable XQuery engines available today. The software is available via `http://www.pathfinder-xquery.org/`; two publications give an overview of the system setup:

[Boncz05c] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: XQuery—The Relational Way. In *Proc. of the 31st Int'l Conference on Very Large Databases (VLDB)*, pages 1322–1325. Trondheim, Norway, September 2005.

[Boncz05a] Peter Boncz, Torsten Grust, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: Relational XQuery over Multi-Gigabyte XML Inputs in Interactive Time. Technical Report INS-E0503, CWI, Amsterdam, March 2005.

By exploiting *memory mapping* facilities in modern operating systems, we extended the MonetDB/XQuery implementation into an XQuery system with full transaction and update support. We demonstrated its practicability as well as the employed *peephole* optimization strategy in

[Boncz06a] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Sjoerd Mullender, Jan Rittinger, and Jens Teubner. MonetDB/XQuery—Consistent & Efficient Updates on the Pre/Post Plane. In *Proc. of the 10th Int'l Conference on Extending Database Technology (EDBT)*. Munich, Germany, March 2006.

[Boncz06b] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of the 2006 SIGMOD Int'l Conference on Management of Data*. Chicago, IL, USA, June 2006.

The peephole optimization strategy as well as other aspects of Pathfinder's query optimizer will be our topic for Chapter 5, which we will conclude with an experimental assessment of the MonetDB/XQuery system. Chapter 6 finally wraps up.

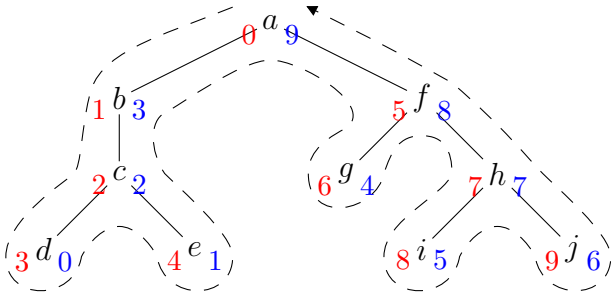
2 Relational XML Storage

An appropriate storage model for its underlying data lies at the very heart of any database implementation. As this thesis focuses on XML processing on *relational* back-ends, whose table-shaped data model seems contrary to the tree-based XML model, our challenge is to find a proper translation that allows for the lossless storage of XML content in relational tables, while enabling the RDBMS to efficiently evaluate XPath location steps.

Our findings are based on a schema-oblivious *tree encoding*. The *XPath accelerator* described by Grust [Grust02] maps each node in the XML tree structure to a binary tuple $\langle pre, post \rangle$. After a short review of the encoding in Section 2.1, we will elaborate on some of its specifics that will aid the efficient processing of XPath. A thorough *experimental assessment* in Section 2.2 highlights strengths and weaknesses of the evaluation strategies employed by existing RDBMS implementations with respect to encoded tree data. We will close this chapter in Section 2.3 with a brief summary of work related to the approach pursued here.

2.1 XPath Accelerator Encoding

While the W3 Consortium specifies XML in terms of the lexical structure of XML files, the file format actually describes a serialized representation of *ordered, unranked trees*. Hence, a proper relational encoding of this underlying data model forms the starting point for any relational XQuery processor. Primarily, such an encoding must allow for the efficient execution of XQuery's document access



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

Figure 2.1: Tree walk to determine preorder ranks $pre(v)$ (left numbers) and postorder ranks $post(v)$ (right numbers) for an example document of ten nodes.

facilities, namely its sublanguage *XPath*.

Our work is based on the *XPath accelerator* encoding proposed by Grust [Grust02]. The encoding keeps information on the structural component of the XML document in a pair of integer values and supports the efficient evaluation of all 12 XPath axes—and thus XQuery's *full axis* feature—, starting from arbitrary context nodes.

2.1.1 Pre- and Postorder Ranks

XPath accelerator annotates all nodes in the XML structure according to their occurrence in a *pre-* and *postorder traversal* of the tree. For any tree node v, we record its pre-/postorder rank in the pair $\langle pre(v), post(v)\rangle$. The encoding is efficient to generate: both values can easily be derived in a single sequential document read, *e.g.*, using a SAX [SAX] parser. Figure 2.1 illustrates this technique for a small example tree.

Once values $pre(v)$ and $post(v)$ have been determined, any XPath axis can directly be mapped to a range condition on these values, where any tree node may serve as the step's context node. For example, for the `descendant` axis we have that

$$\begin{gathered} v' \in v\texttt{/descendant} \\ \Leftrightarrow \\ pre(v) < pre(v') \ \wedge \ post(v') < post(v)\,. \end{gathered} \tag{2.1}$$

An intuitive explanation for this condition can be derived from the serialized XML representation. Pre- and postorder ranks $pre(v)/post(v)$ describe the order in which element start and end tags are encountered in a sequential document read, respectively. So Condition 2.1 may be read as: v' is a descendant of v if its start tag <v'> is read after <v> and </v'> occurs before the closing tag </v> in the XML stream. ("Element v' is contained in v.")

Axis α	Axis Predicate $axis(\alpha, v, v')$
`ancestor`	$pre(v') < pre(v) \wedge post(v') > post(v) \wedge kind(v') \neq \mathsf{attr}$
`ancestor-or-self`	$pre(v') \leq pre(v) \wedge post(v') \geq post(v) \wedge kind(v') \neq \mathsf{attr}$
`child`	$par(v') = pre(v) \wedge kind(v') \neq \mathsf{attr}$
`descendant`	$pre(v') > pre(v) \wedge post(v') < post(v) \wedge kind(v') \neq \mathsf{attr}$
`descendant-or-self`	$pre(v') \geq pre(v) \wedge post(v') \leq post(v) \wedge kind(v') \neq \mathsf{attr}$
`following`	$pre(v') > pre(v) \wedge post(v') > post(v) \wedge kind(v') \neq \mathsf{attr}$
`following-sibling`	$pre(v') > pre(v) \wedge par(v') = par(v) \wedge kind(v') \neq \mathsf{attr}$
`preceding`	$pre(v') < pre(v) \wedge post(v') < post(v) \wedge kind(v') \neq \mathsf{attr}$
`preceding-sibling`	$pre(v') < pre(v) \wedge par(v') = par(v) \wedge kind(v') \neq \mathsf{attr}$
`parent`	$pre(v') = par(v) \wedge kind(v') \neq \mathsf{attr}$
`self`	$pre(v') = pre(v) \wedge kind(v') \neq \mathsf{attr}$
`attribute`	$par(v') = pre(v) \wedge kind(v') = \mathsf{attr}$

Table 2.1: XPath axes α and their associated axis predicate $axis(\alpha, v, v')$ (context node v).

Observe that the preorder rank $pre(v)$ trivially implements XQuery's *node identity* operator `is`. As it coincides with the XML document order, we will also use it to express XQuery operators that test for document order (<</>>).

2.1.2 XPath Axis Conditions

If stored in an RDBMS table together with each node's semantic content (*e.g.*, its tag name) such as suggested in Figure 2.1, values $pre(v)$ and $post(v)$ constitute a slim relational encoding of XML document trees. Grust [Grust02, Grust04e] maintains semantic node content in the two columns $kind(v)$ and $prop(v)$, the former encoding each node's *kind* as one of the values elem, attr, text, comm, doc, or pi. The latter contains the tag name for element or attribute nodes and textual node content for the remaining node kinds.

For efficient and easy characterization of the non-recursive XPath axes, Grust adds a fifth column $par(v)$ to hold the preorder rank of each node's parent. In this five-column schema $\langle pre, post, par, kind, prop \rangle$, the predicate $axis(\alpha, v, v')$ characterizes the result set v' of XPath axis α, as seen from context node v. $axis(\alpha, v, v')$ is lined up for the 12 XPath axes in Table 2.1. The conjunctive predicate $test(\nu, v')$ accounts for a node test ν in a step $\alpha::\nu$ (Table 2.2). Both predicates are efficiently implementable, *e.g.*, on SQL hosts.

Node test ν	Predicate $test(\nu, v')$
`node()`	*true*
`text()`	$kind(v') = \mathsf{text}$
`comment()`	$kind(v') = \mathsf{comm}$
`*`	$kind(v') \in \{\mathsf{elem}, \mathsf{attr}\}$
n (name test)	$kind(v') \in \{\mathsf{elem}, \mathsf{attr}\} \wedge prop(v') = n$

Table 2.2: Predicate $test(\nu, v')$ to evaluate node test ν in XPath location step $\alpha::\nu$ (selection).

2.1.3 Illustrating XPath Accelerator: The *pre/post* Plane

In the following, we will use a more illustrative representation of the ranks $pre(v)$ and $post(v)$. In Figure 2.2(a), we have used both values as coordinates to map the nodes of our example document into the two-dimensional *pre/post plane*. The dotted lines indicate that the numbering scheme actually encodes the full tree structure. The adjacent Figure 2.2(b) replicates that tree structure for comparison.

In our new representation, the *pre/post* plane, axis predicates $axis(\alpha, v, v')$ correspond to *region conditions*. The shaded regions in Figure 2.2 illustrate the axis conditions $axis(\alpha, v, v')$ for the four XPath axes `ancestor`, `descendant`, `following`, and `preceding`, as seen from context node f. These four axes *partition* the entire *pre/post* plane into four disjoint regions: nodes in the quadrants top-left, top-right, bottom-left, and bottom-right (as seen from f) constitute f's ancestor, following, preceding, and descendant nodes, respectively.

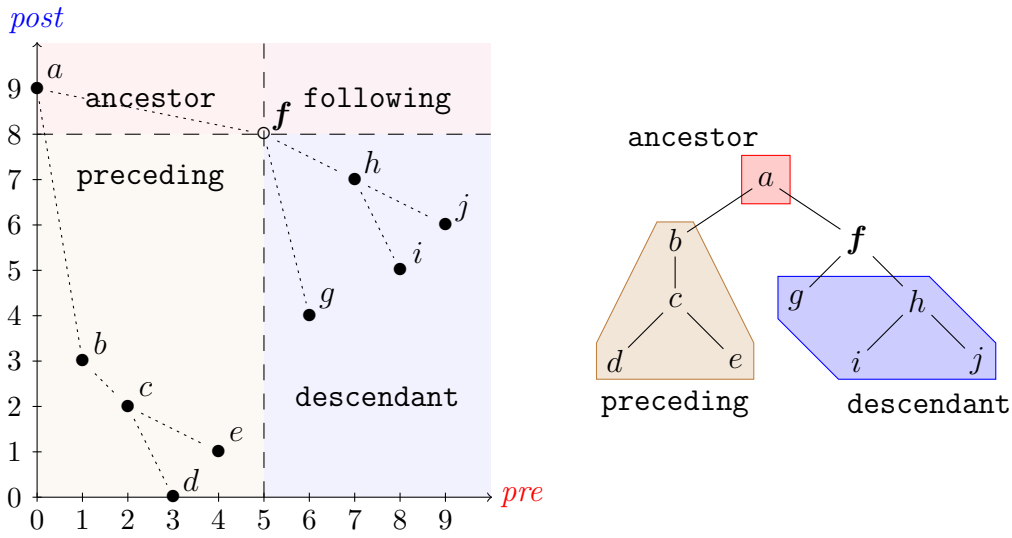
This partitioning property holds for *any* context node in the document as a direct consequence of the axes' definition in XPath. For any context node v, the set

$$v\texttt{/ancestor} \cup v\texttt{/descendant} \cup v\texttt{/following} \cup v\texttt{/preceding} \cup \{v\}$$

contains each document node exactly once. From now on, we will thus summarize these axes under the term *major axes*.

2.1.4 SQL-Based XPath Evaluation

Based on the two predicates $axis(\alpha, v, v')$ and $test(\nu, v')$, XPath location paths may straightforwardly be translated into query specifications for a relational database back-end. The query template $sql(e/\alpha::\nu)$ defines such a translation to SQL in a



(a) *pre*/*post* plane. (b) Regions for major XPath axes.

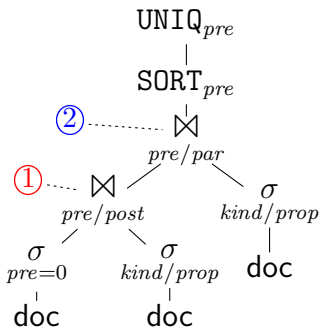
Figure 2.2: The *pre*/*post* plane (left) illustrates XPath axis conditions for the four major XPath axes `ancestor`, `descendant`, `following`, and `preceding` as seen from node f. Corresponding tree regions are shown on the right.

fully compositional manner:

$$sql(e/\alpha::\nu) \equiv \begin{array}{l} \texttt{SELECT DISTINCT } v'.* \\ \texttt{\ \ FROM } sql(e) \texttt{ AS } v, \texttt{ doc AS } v' \\ \texttt{\ WHERE } axis(\alpha, v, v') \\ \texttt{\ \ \ AND } test(\nu, v') \\ \texttt{ORDER BY } v'.pre \ . \end{array} \tag{2.2}$$

$sql(e/\alpha::\nu)$ selects the tuples, *i.e.*, nodes, in `doc` that are reachable from context set e via location step $\alpha::\nu$. Steps may start at any context set e, represented as the subset $sql(e)$ of the document relation `doc`. Specifically, $sql(e)$ may be the result of a previous location step, or—for *absolute* path expressions (starting with `/`)—the document root, *i.e.*, the tuple with $pre(v) = 0$. Clauses `DISTINCT` and `ORDER BY` ensure XPath's semantics of a duplicate-free result, sorted in document order.

This XPath-to-SQL compilation scheme leads to queries of nesting depth k for paths of length k that can be straightforwardly unnested. The XPath expression `/descendant::city/following-sibling::zipcode`, for instance, can be mapped to the SQL expression listed in Figure 2.3(b). Effectively, this translation turns XPath expressions of k location steps into a k-fold self-join of the document relation



(a) Relational plan.

```
SELECT DISTINCT z.*
  FROM doc AS d, doc AS c, doc AS z
 WHERE d.pre = 0
   AND c.pre > d.pre AND c.post < d.post     (1)
   AND c.kind = elem AND c.prop = 'city'
   AND z.pre > d.pre AND z.par = d.par       (2)
   AND z.kind = elem AND z.prop = 'zipcode'
 ORDER BY z.pre
```

(b) SQL equivalent obtained using $sql(e/\alpha::\nu)$.

Figure 2.3: Query template $sql(e/\alpha::\nu)$ translates k-step XPath expressions into a k-fold self-join of relation doc, with join predicates corresponding to the respective step predicates (query `/descendant::city/following-sibling::zipcode`).

doc, where axis conditions serve as join predicates over *pre* and *post* ranges. A possible evaluation strategy could be the query plan shown in Figure 2.3(a).

2.1.5 Index Support for XPath Accelerator

The self-joins employed in the relational XPath evaluation plans of the previous section describe axis conditions in terms of *region conditions* on the two-dimensional *pre*/*post* plane. *Multi-dimensional* indexing techniques may, hence, provide efficient support to evaluate location steps over encoded tree data.

Among the well-known index structures for multi-dimensional data (*e.g.*, Grid Files [Nievergelt84], Quad Trees [Finkel74]), the value distribution of *pre*/*post*-encoded XML data suggests the use of *R-trees* [Guttman84]. They are known to adapt well to non-uniform point distributions in a low-dimensional space. Figure 2.4 illustrates this distribution for an example document of 220 nodes.

The effectiveness of R-tree indexing for relational XPath evaluation has, in fact, been confirmed by Grust's experimental studies in [Grust02]. R-trees, however, have hardly found their way into mainstream RDBMS products, where B-trees are still the predominant means to index data. A workaround is the use of *concatenated* B-trees. But as Grust points out, B-trees may be used to scan the *pre*/*post* plane along only one direction at a time. Regardless of the choice of this direction (*pre* or *post*), the system is doomed to encounter a large number of *false hits* during the scan.

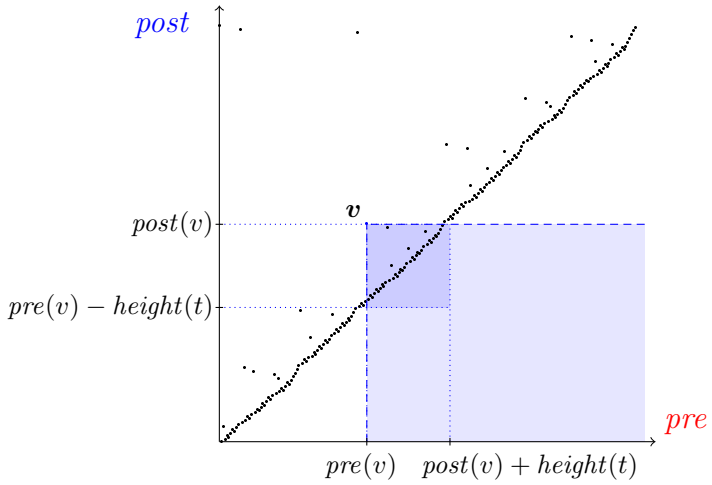


Figure 2.4: Node distribution in the *pre*/*post* plane for an example document of 220 nodes. Original (----) and shrunken (·····) scan ranges for a `descendant` query rooted at v.

2.1.6 Techniques to Reduce the Search Space

Regardless of the choice of an indexing technique, the determining cost factor for query evaluation in the *pre*/*post* plane is the *size* of the query window implied by the axis join predicate $axis(\alpha, v, v')$. In Figure 2.4, this query window covers almost 1/4 of the plane, though only few nodes qualify as descendants of v—the database scans a large amount of unoccupied space.

A node distribution as in Figure 2.4 is typical for XML documents of realistic size: for larger documents, tree nodes accumulate along a diagonal line; only few nodes are located in the upper-left region, while the bottom-right region is entirely empty. It is promising to exploit this observation and provide the database system with tighter bounds for XPath query regions and, hence, reduce the involved search space for each step.

A fundamental correlation to derive such bounds has already been observed by Grust [Grust02]. For any tree node v, its pre- and postorder ranks $pre(v)/post(v)$ relate according to

$$pre(v) - post(v) = \underbrace{level(v)}_{\leq height(t)} - size(v)\,, \tag{2.3}$$

where $level(v)$ describes v's distance from the tree root, $size(v)$ is the number of nodes in v's `descendant` region, and $height(t)$ the length of the longest root-to-leaf path in the tree. This correlation is a direct consequence of the fact that the involved properties represent the relational encoding of a *tree*. For a concise

derivation of the equation we refer the reader to, *e.g.*, [Rode03].

In case we do not have a node's *level* information at hand, the overall tree height $height(t)$ may serve as a reasonable over-estimation for $level(v)$. In practice, we have found $height(t) \lesssim 15$ even for large XML instances. Hence, the estimation error is negligible in comparison the total document size.

Shrink-Wrapping the `descendant` Axis

Correlation 2.3 forms the basis for the so-called *shrink-wrapping* technique described by Grust [Grust02], an optimization that may significantly reduce the search space for the `descendant` axis. Nodes in the `descendant` region of any node v are consecutively assigned preorder ranks after v itself, hence,

$$v' \in v/\texttt{descendant} \;\Rightarrow\; pre(v') \leq pre(v) + size(v) \;\; . \tag{2.4}$$

With a similar reasoning on postorder ranks and with Equation 2.3, this inspires the introduction of additional *shrink-wrapping constraints* for the axis predicate $axis(\texttt{descendant}, v, v')$:

$$pre(v') \;\leq\; post(v) + height(t) \;\text{ and} \tag{2.5a}$$
$$post(v') \;\geq\; pre(v) - height(t) \;\; . \tag{2.5b}$$

As illustrated in our example in Figure 2.4, this may significantly reduce the size of the associated query window. The shrink-wrapping conditions overestimate the actual `descendant` area by at most $height(t)$ (which we found small even for large XML instances). Moreover, this reduction makes the size of the query window *independent* of the total document size. We will observe an increase in performance of several orders of magnitude as the result of shrink-wrapping in the experimental section of this chapter.

Stretching the *pre*/*post* Plane

We may avoid the problem of false hits in a B-tree-indexed plane altogether, if we apply minor modifications to the document encoding itself. The reason for these false hits is the specification of query windows based on two *independent* range predicates (on dimensions *pre* and *post*). The scan along either dimension (using a B-tree) leads to numerous false hits with regard to the other.

An important aspect of the axis predicates in Table 2.1 is that they define *pre*/*post* query windows relative to $pre(v)$ and $post(v)$, *i.e.*, independent of their absolute values. We can exploit this observation by slightly modifying the computation of $pre(v)$ and $post(v)$: *Couple* preorder and postorder ranks such that whenever *pre* is incremented, *post* is as well and vice versa.

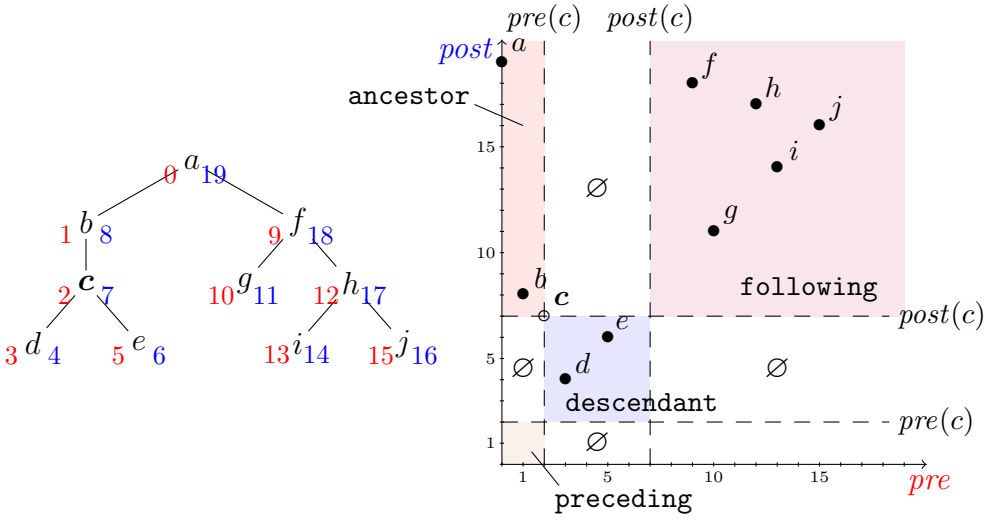


Figure 2.5: Stretched preorder/postorder rank assignment and resulting *pre*/*post* plane. Axes `preceding`, `descendant`, and `following` are sufficiently characterized as ranges along a *single* dimension only (*pre* or *post*).

The resulting *stretched* *pre*/*post* plane is illustrated in Figure 2.5. For any node $v' \in v/\texttt{descendant}$, we now have that

$$pre(v) < pre(v') < post(v) \qquad \text{as well as} \qquad pre(v) < post(v') < post(v) \ .$$

No other nodes in the plane can fulfill this property, since we incremented *pre* and *post* monotonically after traversing the subtree of v, *i.e.*, the regions marked by $\varnothing$ in Figure 2.5 are necessarily empty. As a consequence, it becomes sufficient to scan the plane along *either* dimension to evaluate the `descendant` axis without encountering any false hits. The same considerations also allow for the characterization of the axes `preceding` and `following` in terms of a one-dimensional range query only.

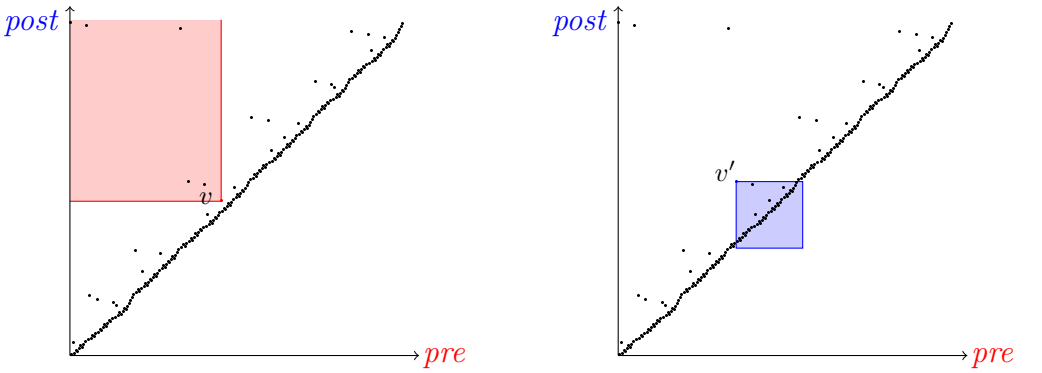
Note that we have not lost any of the other valuable properties of the *pre*/*post* plane:

(i) predicates $axis(\alpha, v, v')$ continue to work as before for all axes,

(ii) $pre(v)$ still implements document order and uniquely identifies document node v, and

(iii) we can now accurately estimate the subtree size below node v:

$$size(v) = \frac{1}{2}\left(post(v) - pre(v) - 1\right) \ .$$



(a) `/descendant::`n`/ancestor::`m. (b) `/desc-or-self::`m`[descendant::`n`]`.

Figure 2.6: Scan region for an `ancestor` step taken from node v (left). Rewriting the query into its symmetric equivalent yields a shrink-wrapped query window (now axis `descendant`) as shown on the right.

As we showed in [Grust04e], the single-dimensional range conditions for the stretched *pre*/*post* plane can lead to an additional 10 % performance increase in comparison to the shrink-wrapping approach. At the same time, the stretched plane no longer requires the explicit maintenance of $height(t)$ as an estimate for $level(v)$. For lack of space, we omit a detailed discussion of the stretched *pre*/*post* encoding here and refer the reader to [Grust04e] for a detailed study.

Symmetries in XPath

Though we have seen that shrink-wrapping and *pre*/*post* plane stretching may significantly reduce the size of scan regions in the *pre*/*post* plane, both optimizations primarily apply to the XPath `descendant` axis. As Olteanu *et al.* [Olteanu02] have observed, *symmetries* between XPath axes may be used to rewrite path expressions into equivalent ones with possibly better execution plans. This turns out to have an unanticipated impact on relational XPath evaluation.

Consider the XPath expression below that retrieves all elements named m which contain at least one element with tag name n:

`/descendant::`n`/ancestor::`m .

This expression taken literally involves searching all n nodes in the *pre*/*post* plane (`/descendant::`n), and then, for each node v of them, a scan of v's `ancestor` region to retrieve result nodes with tag name m. The scan region for the latter step is depicted in Figure 2.6(a).

```
   SELECT DISTINCT v_m.*
     FROM doc AS v_r, doc AS v_n,
          doc AS v_m
1   WHERE v_r.pre = 0
2     AND v_n.pre > v_r.pre
3     AND v_n.post < v_r.post
4     AND v_n.prop = n
5     AND v_m.pre < v_n.pre
6     AND v_m.post > v_n.post
7     AND v_m.prop = m
    ORDER BY v_m.pre
```

(a) `/descendant::`n`/ancestor::`m.

```
   SELECT DISTINCT v_m.*
     FROM doc AS v_r, doc AS v_m,
          doc AS v_n
1   WHERE v_r.pre = 0
2     AND v_m.pre ≥ v_r.pre
3     AND v_m.post ≤ v_r.post
4     AND v_m.prop = m
5     AND v_n.pre > v_m.pre
6     AND v_n.post < v_m.post
7     AND v_n.prop = n
    ORDER BY v_m.pre
```

(b) `/desc-or-self::`m `[descendant::`n`]`.

Figure 2.7: Corresponding SQL code for a pair of symmetric XPath expressions. The two SQL queries may serve as a proof for the equivalence of the two paths.

If, instead, we first rewrote the expression according to [Olteanu02], we could trade the `ancestor` for an XPath `descendant` step:

$$\texttt{/descendant-or-self::}m\,\texttt{[descendant::}n\texttt{]}\ .$$

In other words, we search the document for each node v' with tag m. Then, for each of them, we evaluate `descendant::`n and reject v' if the XPath predicate result is empty.

Under these circumstances, the query becomes a candidate for applying the shrink-wrapping technique which leads to a significant scan size reduction. This benefit is clearly recognizable in Figure 2.6(b) which illustrates the resulting query window for the second evaluation step. Remember that shrink-wrapping leads to query window sizes that are *independent* of the overall document size. The rewrite will be even more effective on larger document instances.

While Olteanu *et al.* argue the correctness of the rewrites based on specific properties of XPath location steps, we may, in fact, prove these equivalences in a purely algebraic fashion on the basis of the XPath accelerator encoding. Figure 2.7 illustrates this with the help of the SQL clauses that represent a symmetric pair of XPath expressions.

Conditions 1 and 4–7 are identical for both queries. As for the remaining Conditions 2 and 3, the `descendant` predicate in Figure 2.7(a) is implied by the transitive combination of lines 2 and 5, and 3 and 6 in Figure 2.7(b), respectively. In the opposite direction, the fact that the root node tuple v_r, by construction,

is assigned the table's overall minimum *pre* and maximum *post* value trivially satisfies Conditions 2 and 3 in Figure 2.7(b).

2.1.7 Range Encoding: An Alternative to *pre/post*

The original XPath accelerator proposal provides a simple, yet efficient tree encoding with values $pre(v)$ and $post(v)$ as the primary carrier of structural information. The *pre/post* plane as its visualization offers an intuitive insight into query processing on such encoded tree data. Specific application needs, however, may drive the decision to alternatives to *pre/post*.

Equation 2.3 relates the four node properties $pre(v)$ (preorder rank of v), $post(v)$ (postorder rank of v), $level(v)$ (v's distance from the tree root), and $size(v)$ (the number of nodes in the subtree below v) in a single equation. As a consequence, we can recover the original *pre/post* values from any encoding that provides at least three of these four values. In that sense, *any* such encoding is equivalent to XPath accelerator. In fact, if an encoding involves at least either $pre(v)$ or $post(v)$, one more value suffices to fully encode a tree.

The Pathfinder compiler, whose foundations we describe in this thesis, as well as the compilation procedure that we pursue in Chapter 4 are based on such an alternative to *pre/post*. The *range encoding* employs the triple $\langle pre(v), size(v), level(v)\rangle$ and turns out to exhibit a set of interesting properties:

(i) Implicit search space minimization.
On *pre/size*-encoded data, the XPath `descendant` axis naturally corresponds to a region scan along a single dimension only:

$$\underset{pre/size}{axis}\,(\texttt{descendant}, v, v') = pre(v) < pre(v') \leq pre(v) + size(v)\ .$$

Needless to say that this scan is efficiently implementable using, *e.g.*, a B-tree index scan.

(ii) Efficient construction of transient nodes.
The *pre/post* encoding leads to extensive *renumbering* costs in case of structural *updates* of the XML tree (see Section 2.1.8). Values $size(v)$ and $level(v)$ are *invariant* with respect to subtree copying or moving, which, as we shall see in Chapter 4, can significantly lower the costs for XQuery's element construction operator.

(iii) Enhanced estimation accuracy.
The maintenance of an explicit $level(v)$ column turns out to significantly enhance the accuracy of cost estimations on encoded XML documents [Rode03]. This may be a promising leverage point for future enhancements of the Pathfinder XQuery system.

In addition, the Pathfinder implementation abandons column $par(v)$ from its relational storage. As already sketched by Rode [Rode03], the efficient implementation of non-recursive XPath axes (`child`, `parent`) may easily be recovered if the respective algorithm is aware of specific tree properties captured by the values $pre(v)$ and $level(v)$ in range-encoded data. In the experimental section of this chapter, we will see that even off-the-shelf RDBMSs adapt gracefully to the missing parent pointer.

A positive side-effect of replacing $par(v)$ with $level(v)$ is its reduced storage consumption: while a single byte typically suffices to store $level(v)$, $par(v)$ has to scale with XML document sizes.

2.1.8 A Word on Updates

Tree nodes are consecutively assigned their pre- and postorder ranks during a tree traversal as described in Section 2.1.1. While this can be efficiently implemented for an initial tree load [Grust04e], the insertion of a new node v comes at a high cost: the preorder and postorder ranks of *all* nodes in v's `following` and `ancestor` regions must be adapted accordingly.[1] To delete a node, however, it suffices to simply remove the corresponding tuple from relation `doc`.

High update costs are an inherent problem of numbering schemes that use a fixed-width encoding [Cohen02]. Nevertheless, the Pathfinder system supports efficient structural updates on encoded XML documents with its *pointer swizzling* technique [Boncz06a].

2.2 XPath on Commodity RDBMSs

To see how off-the-shelf database implementations can cope with *pre/post*-encoded data, we created several instances of a *pre/post* table and fed them into two popular RDBMSs: Version 8.2 of IBM's DB2 Universal Database (Enterprise Edition) as one of the big players in the database industry and PostgreSQL 7.3.3 as a complete open-source implementation of SQL. Both back-ends had been installed on a SuSE Linux Enterprise Server 9 system, equipped with 2×3.2 GHz Intel Xeon processors and 8 GB RAM. The system was running off four SCSI hard drives (140 GB each, 10,000 rpm), two of which were dedicated to DB2.

The initial nine XML document instances were generated using the XML document generator `xmlgen` from the XMark benchmark project [Schmidt02]. Their sizes ranged between 113 KB and 1.1 GB (5,256 to 50,844,982 tree nodes, respectively). The benchmark documents model an internet auction site and use the

[1] Note that not only the update overhead is causing trouble here. Updating all ancestor nodes of v also imposes a significant *locking bottleneck*.

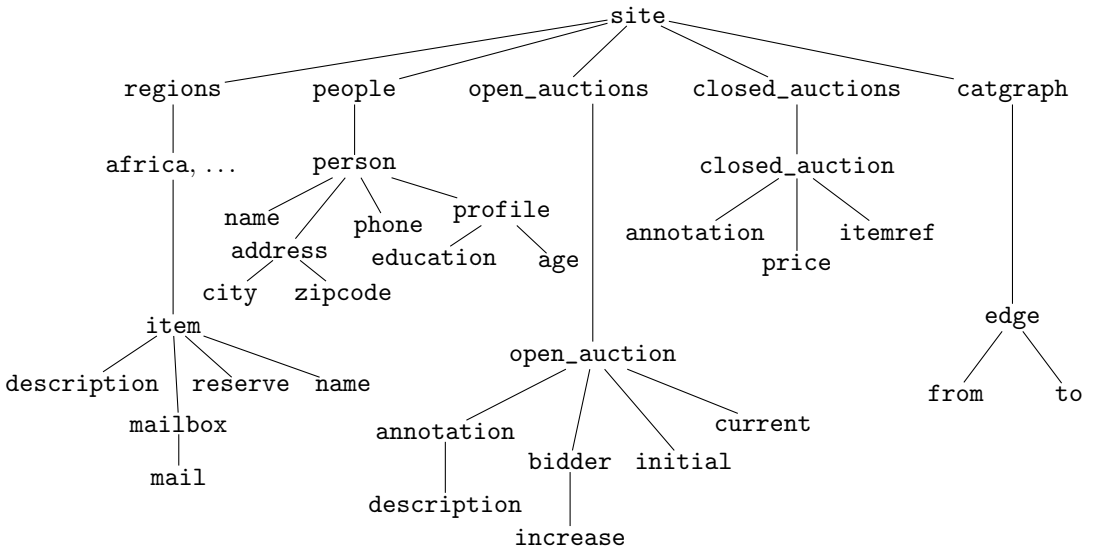


Figure 2.8: Element hierarchy of XMark document instances (excerpt).

schema sketched in Figure 2.8. `xmlgen` scales generated documents very regularly, so that result sizes for XPath expressions typically grow linearly with the document size.

Before loading them into the database systems, the generated documents were not only *pre*/*post*-, but also range-encoded (*pre*/*size*/*level*) for comparison with the storage scheme of the Pathfinder system.

Indexes Created. The DB2 database system ships with a sophisticated index wizard that suggests a set of indexes based on a typical database query workload. We prepared a workload of different SQL queries that represent a selection of various XPath expressions and set up indexes as suggested by the advisor. The suggestions of the advisor included indexes on the structural component of our storage (*pre* and *post*) as well as on the remaining node properties.

On the PostgreSQL installation, we created a combined $\langle pre, post, kind, prop\rangle$ B-tree index for the efficient evaluation of structural constraints. The inclusion of columns *kind* and *prop* allows for the evaluation of name tests within index scans. An additional index on $\langle parent, pre, kind, prop\rangle$ backs the execution of non-recursive axes.

Query Set. Against all XMark instances, we ran a selection of XPath queries each of which stresses a specific aspect of the XPath accelerator encoding:

- Q_1: `/descendant::open_auction/descendant::description`

Query	0.11 MB	1.1 MB	11 MB	111 MB	1,118 MB
Q_1	12	120	1,200	12,000	120,000
Q_2	8	77	631	6,409	64,463
Q_3	12	120	1,200	12,000	120,000
Q_4	12	125	1,255	12,716	127,315
Q_5	60	708	6,182	59,486	597,777

Table 2.3: Number of nodes returned by Queries Q_1–Q_5 on different XML document instances. Results grow linearly to document sizes.

- Q_2: `/descendant::age/ancestor::person`
- Q_3: `/descendant::current/preceding::initial`
- Q_4: `/descendant::city/following::zipcode`
- Q_5: `/descendant::open_auction/child::bidder/child::increase`

All queries contain an initial `descendant` step, whose sole purpose is to provide a context set of reasonable size for the following steps that are of actual interest. Queries Q_1 through Q_4 use location steps with recursive semantics, hence, are among the obvious strengths of the XPath accelerator encoding. Q_5, in contrast, examines the support for the non-recursive `child` axis, probably the most important axis in XPath. All queries return result sets that depend linearly on the XMark instance sizes (cf. Table 2.3).

We expressed all five queries in SQL, strictly following the translation rules described in Section 2.1.4. Queries were ran multiple times to determine the average afterwards (though variations in execution time were remarkably small).

2.2.1 DB2 Runs XPath

As one of the major commercial database products, DB2 provides a highly scalable relational platform. In this section, we want to assess how well an off-the-shelf DBMS can cope with tree queries on *pre*/*post*-encoded data.

Querying the `descendant` Region

Figure 2.9 visualizes the query execution times observed for Query Q_1 on the DB2 installation. The system had no problems processing the query on any XML document size loaded. Apart from that, in all cases, the execution cost is dominated by the second location step. (In additional experiments, we ran the query `/descendant::open_auction` on all nine XML instances. Execution times for this

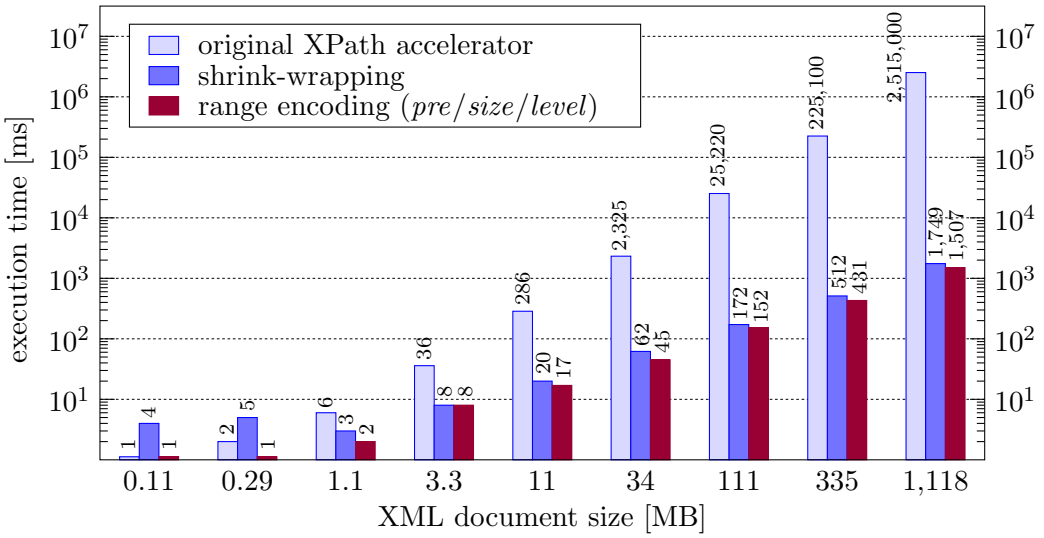


Figure 2.9: XPath evaluation performance for Query Q_1. Shrink-wrapping avoids the quadratic increase observed for the original *pre*/*post* encoding and leads to a linear scaling. The range encoding exhibits a similar behavior.

step grew linearly with the XML document size, but remained negligible compared to the execution times in Figure 2.9.)

The initial `descendant` step in Query Q_1 produces a context set that scales linearly to the number of tuples N in the document table. For each of these context nodes, the scan region for the subsequent `descendant` step also grows linearly to the size of the *pre*/*post* plane, which brings in another factor of N. This leads to a quadratic scaling for the original XPath accelerator implementation as can be seen in Figure 2.9.

The shrink-wrapped query regions for the `descendant` axis overestimate their result set by at most the overall tree height $height(t)$ (Equations 2.5), negligible for large document instances. Therefore, the effort to evaluate the shrink-wrapped variant of Query Q_1 solely depends on the axis step *result*. For documents generated with `xmlgen`, result sizes grow linearly to the XML document size, hence, we see a linear dependency in Figure 2.9.

For comparison, Figure 2.9 also includes the execution times we observed for the equivalent queries on range-encoded data (cf. Section 2.1.7). In this encoding, the `descendant` axis can be described as a one-dimensional range only—ideally supported by a B-tree index scan. Moreover, there is no more over-estimation as in the shrink-wrapping case: the range boundaries are now *exact*. In combination with the computationally less expensive predicate, this leads to a performance improvement of roughly 15 % over the shrink-wrapped *pre*/*post* variant.

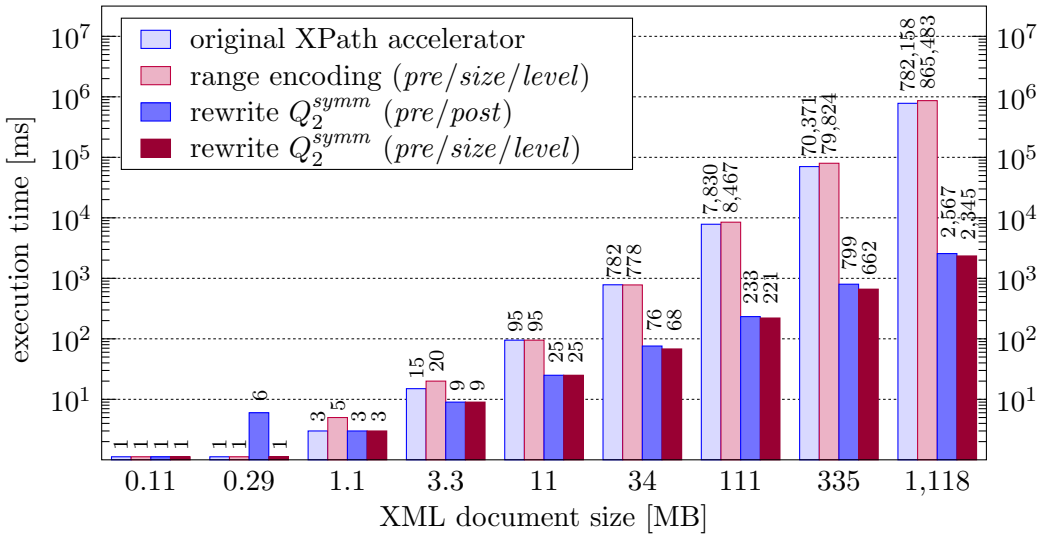


Figure 2.10: XPath performance for query Q_2. The symmetric rewrite of Q_2 according to [Olteanu02] reduces its quadratic complexity to a linear scaling for both encoding variants, *pre/post* as well as *pre/size/level*.

Exploiting XPath Symmetries to Evaluate `ancestor`

The XPath performance for the second query of our test set, Q_2, is lined up in Figure 2.10. Again, a linearly growing context set in combination with the linear growth of the *pre/post* scan regions leads to a quadratic complexity of the respective SQL query. This time, the explicit maintenance of subtree sizes in the range encoding is no help either: the original encoding and its variant are almost at par in Figure 2.10.

Query Q_2, however, is an instance of the query pattern we saw in Section 2.1.6 that allows for a path rewrite according to the symmetry observation of Olteanu *et al.*:

$$Q_2^{symm} = \texttt{/descendant-or-self::person[descendant::age]} \ .$$

On our evaluation platform, the reformulated query exhibits a linear behavior for both encoding variants.[2] Both of them perform equally well, with a slight tendency towards the *pre/size/level*-based encoding.

DB2 Captures XPath Semantics. Rewriting simple XPath expressions according to [Olteanu02] typically leads to the introduction of predicate expressions,

[2] Again, the evaluation on the *pre/post* back-end benefits from shrink-wrapping.

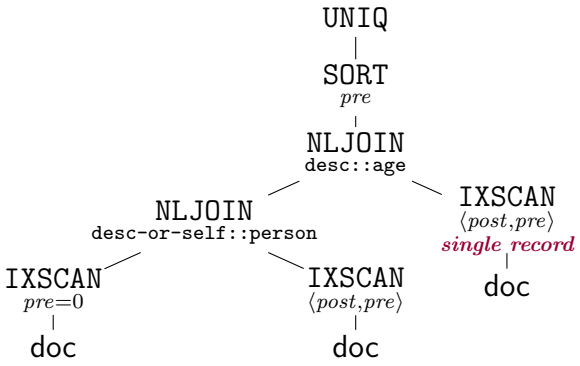


Figure 2.11: The execution plan employed by DB2 to evaluate Query Q_2 directly reflects XPath's existential predicate semantics. Relation `doc` is accessed with a *single record* scan to implement the predicate.

which is also the case for query Q_2^{symm}. XPath specifications prescribe *existential semantics* for these predicates, *i.e.*, systems are free to abort processing for the current context node, as soon as they find the first match for the predicate expression.

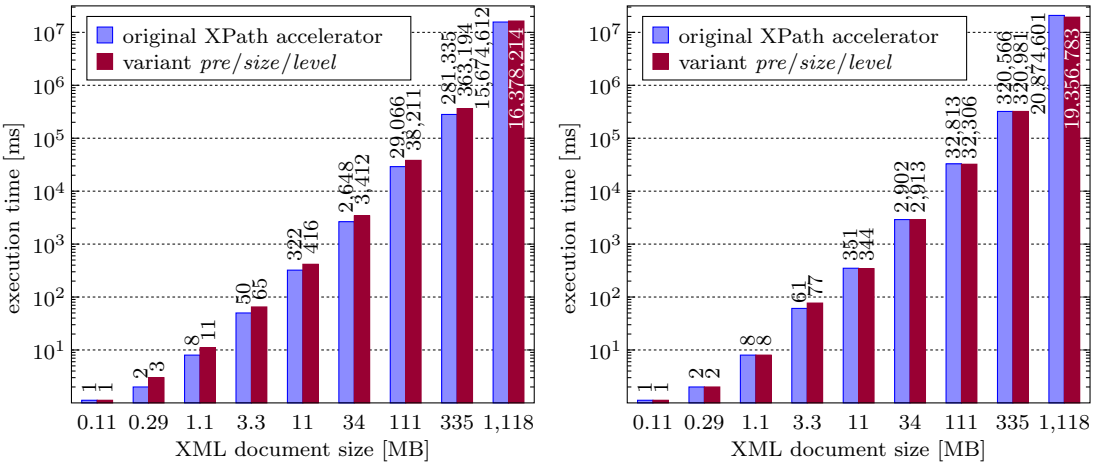
This is exactly what DB2 does. In the query plan that DB2 uses to evaluate Q_2 (see Figure 2.11), we can see how its optimizer seizes the opportunity and executes the search for `age` nodes as a *single record* index scan . With that means, DB2 precisely implements XPath's *early out semantics*.

Axes `preceding` and `following`

Without the application of specific query rewrites, the XPath `descendant` and `ancestor` axes (Queries Q_1 and Q_2, respectively) have shown a quadratic runtime behavior for the examined XMark document instances. We expect the same behavior to apply to the remaining two major XPath axes, `preceding` and `following`. Indeed, we see a quadratic scaling for test queries Q_3 and Q_4 in Figure 2.12; execution times for the 1 GB instance are even worse than quadratic complexity.

In the earlier cases, we were able to achieve linear runtime complexity with specific adaptions to the generated SQL queries (shrink-wrapping) as well as to the path expression itself (symmetric rewrites). None of these optimizations, however, are able to achieve the same effect for the `preceding` and `following` axes. In Chapter 3, we will discover how the introduction of *tree-awareness* can achieve a linear scalability for both axes nevertheless.

Both queries reveal an additional aspect crucial for the scalability of any XPath engine: the overhead incurred by XPath's requirement for a duplicate-free, ordered result. While all queries evaluated so far showed a linear scaling not only for the



(a) Query Q_3 (`preceding` axis). (b) Query Q_4 (`following` axis).

Figure 2.12: Axes `preceding` and `following` lead to a quadratic runtime complexity for XML document instances generated with `xmlgen`.

final, but also for *intermediate* result sizes, the evaluation of Q_3 and Q_4 produces a high volume of duplicates that the database needs to sort and remove afterwards. For documents generated with `xmlgen`, the amount of these duplicates grows quadratically for Queries Q_3 and Q_4, adding up to 7.2×10^9 and 8.1×10^9 tuples on the 1 GB instance. From this document size onwards, we have to pay the toll for the growing *sorting overhead*: DB2 falls significantly behind quadratic scaling for both queries.

The Non-Recursive Case: Evaluation of the `child` Axis

The encoding of the XML document structure per values $pre(v)$ and $post(v)$ has been particularly crafted to accelerate the evaluation of navigation steps with a recursive definition in XPath. Obviously, this should not negatively affect the execution of *non-recursive* axes, namely the axes `child` and `parent`.

For most efficient support of these two axes, Grust makes the parent/child relationship among tree nodes explicit to the RDBMS in terms of the key reference in column $par(v)$ [Grust02]. With an index on $par(v)$, this renders the evaluation of the `child` axis into one of the operations most efficiently implemented in today's database systems: a relational join over index columns. Query Q_5 exploits this operation with two `child` steps and it is not surprising to see that the XPath accelerator execution times in Figure 2.13 scale linearly to our document sizes.[3]

[3]Remember that the linear scaling factor is due to the linearly growing *result* sizes, while the

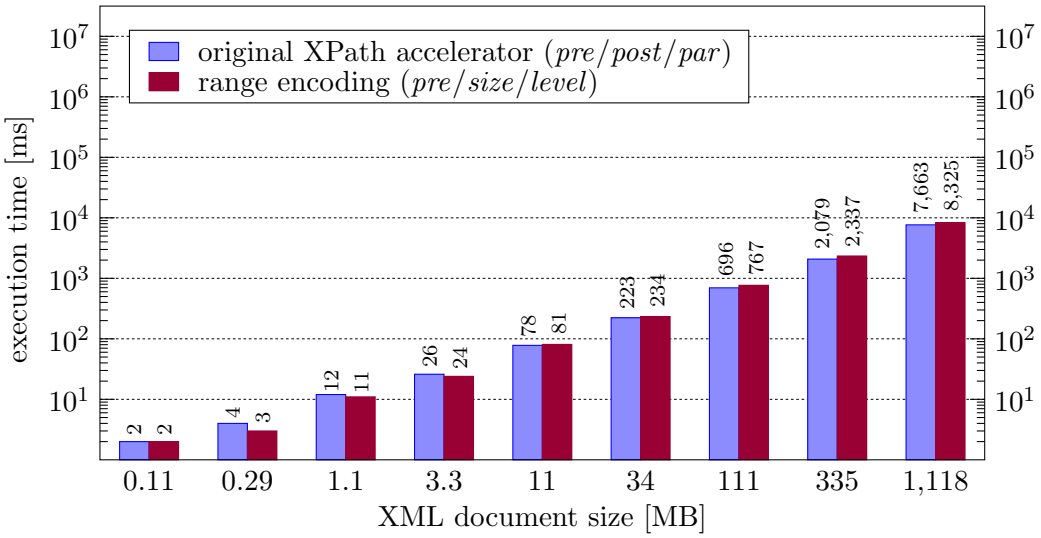


Figure 2.13: Execution times observed for Query Q_5. The evaluation of axis `child` via the combination *pre*/*size*/*level* is almost at par with the foreign key reference in the original *pre*/*post*/*par* encoding.

When introducing the range encoding variant (Section 2.1.7), we abandoned the explicit $par(v)$ column in favor of each node's *level* information, $level(v)$. The result set of a step v`/child` may then be described as the subset of v`/descendant` that belongs to the tree level directly below v, $level(v) + 1$:

$$v' \in v\texttt{/child} \quad \Leftrightarrow \quad v' \in v\texttt{/descendant} \quad \wedge \quad level(v') = level(v) + 1$$

This characterization of the `child` axis seems quite expensive at first, given that the size of v`/descendant` is usually large. Nevertheless, we see only a negligible performance penalty in comparison to an evaluation of the `child` axis via $par(v)$ (Figure 2.13).

Favorable Index Choice. The favorable `child` performance is a consequence of the efficient usage of B-trees to access nodes in the *pre*/*size*/*level* space. An index on the concatenation $\langle level, pre \rangle$ (with the major ordering on column *level*) simplifies the query for a `child` step into the *single* range scan:

$$v' \in v\texttt{/child} \quad \Leftrightarrow \quad \big\langle level(v) + 1, pre(v) \big\rangle < \big\langle level(v'), pre(v') \big\rangle \leq \big\langle level(v) + 1, pre(v) + size(v) \big\rangle \ .$$

execution time of a single `child` step is actually *independent* of the XML document size.

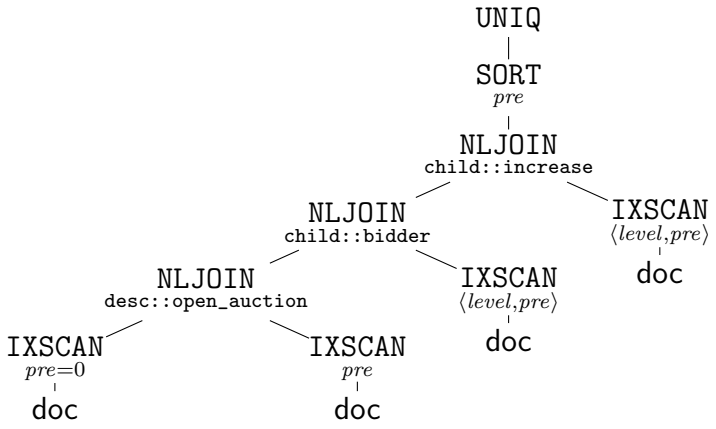


Figure 2.14: Execution plan for Query Q_5 with efficient B-tree usage.

Observe that, in this way, we obtain all children of v in *pre*-order (*i.e.*, document order), without encountering any false hits. The query plan shown in Figure 2.14 uses a B-tree on table `doc` in such a manner to map each `child` step to an index scan along a $\langle level, pre \rangle$ range.

In fact, the plan is almost identical to the one employed by our DB2 instance to evaluate Query Q_5. DB2, however, prepends each index with information on columns *kind* and *prop*. This effectively pushes the evaluation of name tests into index conditions and avoids that index scans access any false tuples from their base relations.

This opportunity to optimize index usage for XPath node tests arises for *all* axes. The DB2 index advisor/optimizer seizes this chance for all queries we evaluated. This constitutes another situation in which the system captures specifics of the *pre*/*post* encoding without any explicit hints or user intervention.

2.2.2 XPath Accelerator on PostgreSQL

Being available in open source, the PostgreSQL database system provides the ideal playground for tree-specific kernel tweaks, such as the *staircase join* algorithm that will be described in the following chapter. Here, we form the baseline for an evaluation of these kernel tweaks by looking at the XPath performance of an *unmodified* PostgreSQL instance.

The trends observed on the PostgreSQL installation precisely reflect those measured on DB2 before. The open-source system lags behind its larger brother by almost an order of magnitude, though. On document instances beyond 111 MB, PostgreSQL did not successfully finish execution within reasonable time ($\approx$ 5 hours) at all.

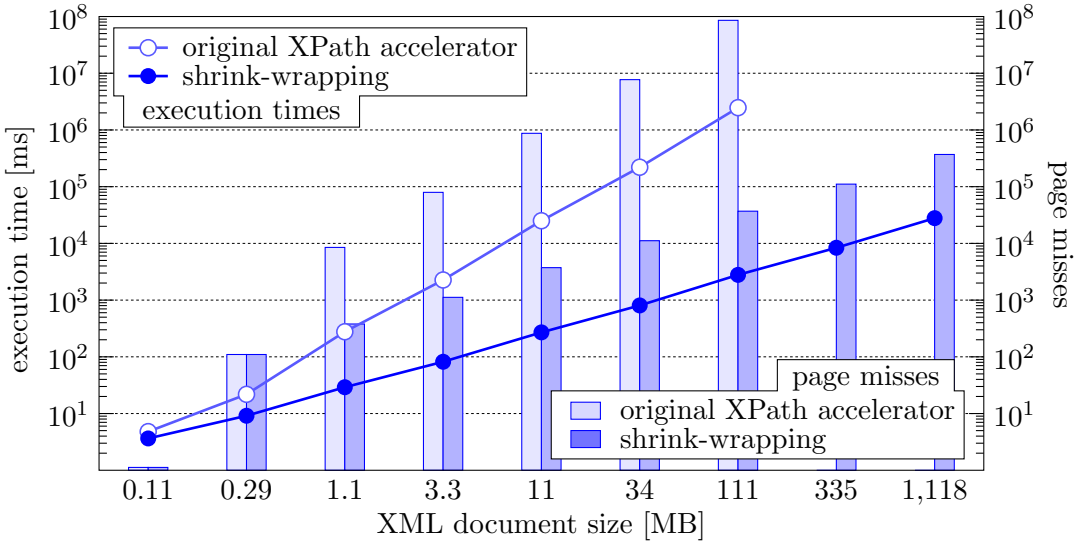


Figure 2.15: Relational evaluation of Q_1 on PostgreSQL. Execution times closely resemble those observed on the DB2 installation before.

The achieved XPath performance for queries Q_1 and Q_2 is illustrated in Figures 2.15 and 2.16, respectively. In addition to that, we used PostgreSQL's facilities to report page access statistics for every query. In both figures, the bar charts depict the total number of *page misses* required to run the query, while the lines represent the overall execution time. (All queries were run with caches "warm"; hence, the small document instances did not require any page loads at all.)

The two measurements confirm the observations we made on our DB2 installation earlier. The tight correlation between query execution times and the amount of page misses indicates scans over large document regions as the major cost factor. As expected, the number of misses grows quadratically to the document size in case of the unaltered XPath accelerator application, while the shrink-wrapping and symmetric rewrite techniques lead to a linear scaling on all XMark instances.

In Chapter 3, we will revisit PostgreSQL, when we investigate the impact of tree-specific enhancements to the kernel itself. At that time, we will also provide performance numbers for PostgreSQL's `preceding` and `following` performance.

2.3 Related Work

The maturity of relational database systems has long since suggested their use as back-ends for XQuery processing and a number of alternative encoding techniques has been published in the literature. In the domain of *schema-oblivious* encodings (cf. Section 1.1.2), these alternatives may be categorized into encodings that use

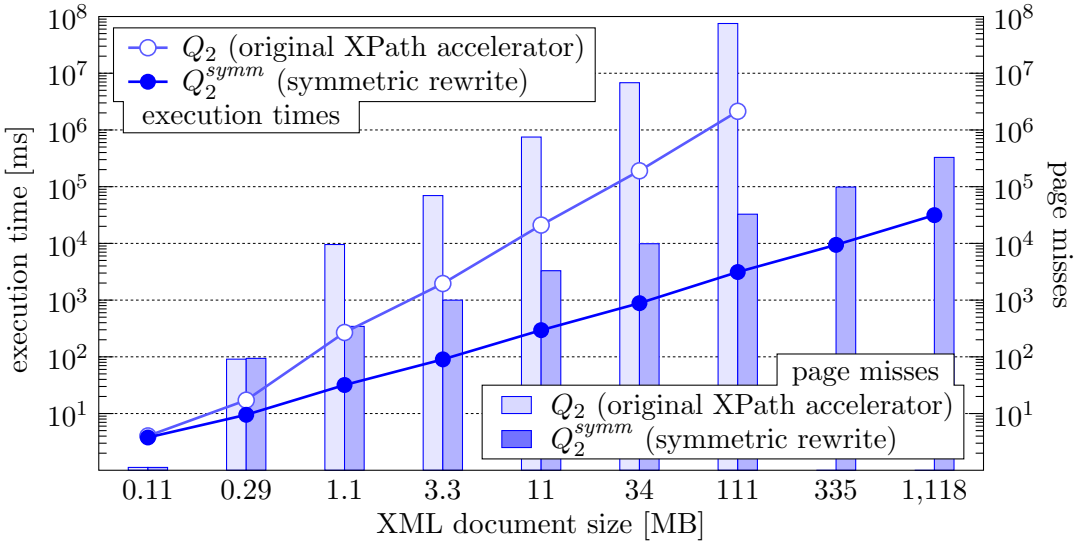


Figure 2.16: Execution times observed for Q_2 on PostgreSQL. Much like DB2, PostgreSQL embraces the cheaper evaluation strategy for the symmetric equivalent, Q_2^{symm}.

a *fixed* number of bits to store the structural part of the XML document tree and *variable-length* encodings.

As we have seen in Section 2.1.8, structural updates require an expensive renumbering of the *pre*/*post* plane. Upon insertion of a new subtree, any node in the `following` and `ancestor` region of the insertion node is subject to such renumbering. Others have suggested to overcome this renumbering problem by the introduction of *gaps* during the determination of the nodes' pre- and post-order ranks [Li01] or the use of floating point numbers to represent $pre(v)$ and $post(v)$ [Amagasa03]. Both approaches, however, do not solve the update problem per se, but can merely defer the renumbering effort, until all gaps are filled up.[4] In fact, Cohen *et al.* proved that for *any* possible labeling scheme, there exists a sequence of N node insertions that requires labels of length $\Omega(N)$ [Cohen02]. Hence, no fixed-length encoding can possibly accommodate updates without relabeling (parts of) its relational storage (and XPath accelerator is no exception).

2.3.1 Fixed-Length Encodings

One of the first relational mappings for XML documents has been described by Florescu and Kossmann [Florescu99]. The *edge mapping* technique models parent-

[4]Note that for a given bit-length of columns $pre(v)$ and $post(v)$, both "solutions" limit the number of nodes that may be accommodated in the doc relation in the first place.

child relationships as directed edges and stores the node identifiers of source and target nodes in two columns of the table *Edge*. Column *ordinal* implements the order among sibling nodes and thus XML document order.

In contrast to XPath accelerator, the edge mapping technique requires recursively defined XPath axes to be evaluated on the *Edge* table in a recursive manner—a processing model that relational databases have not originally been designed for. In return, edge mapping allows for simple and cheap structural updates, where only few nodes require renumbering.

Li and Moon [Li01] describe a relational encoding that closely resembles our range encoding. It uses an *extended preorder* rank plus the node's subtree size to encode the XML structure. However, their work exclusively focuses on the efficient evaluation of the XPath `child` and `descendant` axes.

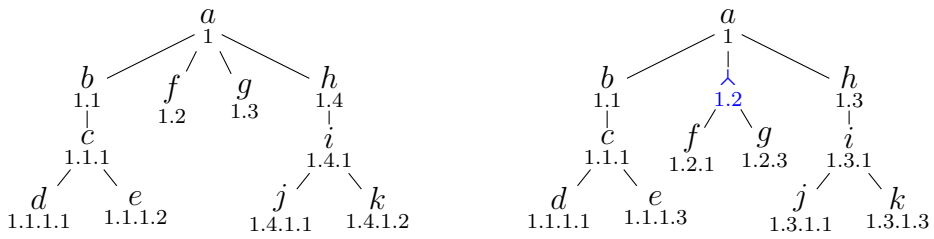
The same is true for the *UID* (unique identifier) encoding that was introduced by Lee *et al.* [Lee96] and later refined by Kha *et al.* [Kha02]. The UID encoding virtually fills up the XML tree structure to obtain a fully populated k-ary tree, where k is the maximum fanout of the original document. The UID of any document node is then its rank in a breadth-first traversal of the virtual tree. Given the UID of a node v, the fixed arity allows to derive all UIDs of v's parent/child nodes. However, given the fact that the fanout of real-world XML instances is rather irregular, with typically few nodes that exhibit a very high fanout, it seems hardly appropriate to base an encoding on the maximum arity in the tree.

A Corner Case: Binary Associations

Somewhat on the borderline to the schema-based storage variant lies the *binary association* technique proposed by Schmidt *et al.* [Schmidt00]. This storage approach is inspired by the edge mapping technique discussed earlier and stores the structural relationship among nodes as explicit parent-child associations.

The technique of Schmidt *et al.* (employed, *e.g.*, in an implementation on top of MonetDB [Boncz02]) *partitions* the XML document according to all root-to-leaf paths found in the overall document (the so-called *path summary*). This naturally leads to a *semantical* grouping of tree nodes into various *association tables*. The authors argue that, depending on the query workload, this data clustering may significantly reduce the amount of processing work spent on irrelevant nodes.

The segmentation of the relational storage along root-to-leaf paths may be beneficial for our *pre/post* storage as well. Apart from that, it could be implemented in a surprisingly simple manner: if we record the root-to-leaf path $path(v)$ for each tree node v, a B-tree index with major ordering on column *path* (*e.g.*, a concatenated $\langle path, pre \rangle$ index) naturally simulates the fragmentation proposed by Schmidt *et al.* in terms of *B-tree partitioning*. This way, we fully benefit from the semantical grouping of binary associations, while we still retain the original XPath



(a) Dewey Order. (b) ORDPATH labels.

Figure 2.17: Variable-length encoding schemes: Dewey Order [Tatarinov02] and its extension ORDPATH [O'Neil04]. The illustration of the latter assumes that f and g were inserted after the initial document load by *careting in* (‸: caret).

accelerator performance (*e.g.*, for wildcard steps). We will see another instance of this unaccustomed use of B-trees in Section 2.3.3.

The addition of $path(v)$ to the relational storage has, in fact, already been proposed by others. Microsoft's SQL Server™, *e.g.*, maintains such information in terms of its *PATH_ID* field [Pal04].

2.3.2 Variable-Length Encodings

Tatarinov *et al.* have early recognized that update costs are an issue in XML data processing [Tatarinov02]. Inspired by the Dewey Decimal Classification[5], the authors propose a numbering scheme that uses a dot-separated sequence of integer numbers to label each node in the tree (cf. Figure 2.17(a)). Labeling according to this *Dewey Order* limits the nodes affected by renumbering to the subtrees in the `following-sibling` region of the insertion point.

Similar to XPath accelerator, Dewey Order labels allow us to efficiently test two nodes u and v for their ancestor/descendant ($u \in v$`/descendant`) or document order relationships ($u \ll v$), solely based on their node labels γ_u and γ_v.

More widely known is a variant of the original Dewey Order encoding, the *ORDPATH* labeling [O'Neil04]. ORDPATH is successfully used in the 2005 version of the Microsoft SQL Server™ and eliminates the need to renumber tree nodes in case of structural updates. The idea is equally simple and effective: during the initial document load, only *positive*, *odd* integer values are used to create ORDPATH labels. Even numbers are reserved as insertion points (*carets*) for later updates. Figure 2.17(b) illustrates the use of carets to incorporate new tree nodes: nodes f and g have been inserted as new children of a between the nodes b and h;

[5] See, *e.g.*, `http://en.wikipedia.org/wiki/Dewey_Decimal_Classification`.

the caret ≺ fills the space between 1.1 and 1.3 and leaves room for even more child additions to a in the future. ORDPATH labels have similar properties to Dewey Order labels, though care has to be taken to ignore carets (easily identified by the even number in their last component) in the case of non-recursive axes (`child`, `parent`, ...).

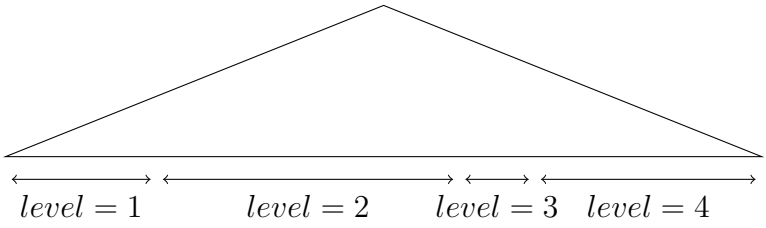
The dotted notation for Dewey-based node labels constitutes an intuitive representation for human reading. Internally, both approaches use a storage format that is inspired by the UTF-8 character encoding [Yergeau03], where the leading bits of the labels determine the bitlength of each integer. In the case of ORDPATH, this leads to a highly compact binary representation.

2.3.3 Relational Database Support

The favorable XPath performance we saw in the foregoing sections is mainly due to the re-use of mature database techniques inherent to the relational systems we examined. Two truly relational techniques are particularly remarkable in our context: the deliberate use of B-trees to index *pre*/*post*-encoded data as well as the exploitation of dense key numberings in modern database kernels.

Index Support for the `child` Axis

We found the evaluation of the `child` axis efficiently supported by prepending the *level* column to an index on preorder ranks. Similarly, columns *kind* and *prop* as leading index keys could significantly accelerate the evaluation of XPath steps that involve name tests. Technically, the prepending of *level* values to an index on preorder ranks effectively leads to a *partitioning* of the resulting B-tree:



This partitioning resembles the work of Graefe on partitioned B-trees [Graefe03]. Graefe proposes the use of *artificial* leading key columns for an enhanced implementation of sort operations in high-volume RDBMSs. He points out that a low-selectivity leading column will typically not hamper query performance, particularly if the underlying B-tree implementation handles prefixes in a proper manner. Prefix B-trees [Bayer77], *e.g.*, provide such an implementation and elegantly compress key prefixes.

Remember that we applied the same "trick" to simulate the effect of the binary association storage model on *pre*/*post*-encoded data in a simple and elegant manner.

The use of a $\langle level, pre \rangle$ index also blends perfectly with the work of Chan *et al.* [Chan04]. They use an artificial *layer axis* to rewrite chains of XPath wildcard steps (*e.g.*, paths of the form e`/child::*/child::*/child::`t) into a single tree navigation. The resulting paths (*e.g.*, e/L^3`::`t) directly translate into conditions on $\langle level, pre \rangle$ ranges and, hence, are ideally backed by the B-tree on $\langle level, pre \rangle$.

Support for Further Properties of the *pre*/*post* Plane

Some database kernels can exploit other interesting properties of the tables generated during the *pre*/*post* encoding of XML trees. The MonetDB system [Boncz02], *e.g.*, provides dedicated support for columns with a *dense key numbering*. MonetDB will only store $pre(v)$ for the first tuple of our encoding and infer remaining *pre* values from the physical tuple order (in terms of its `void` column type). This does not only avoid the materialization of column *pre* (and, hence, reduce storage consumption), but also allows for instantaneous, *positional* access by preorder ranks.

Additional benefits may be gained from a relational database implementation, if we provide its kernel with specific information on the tables' underlying *tree origin*. Such kernel modifications constitute our agenda for the upcoming chapter.

3

XPath Evaluation with Staircase Join

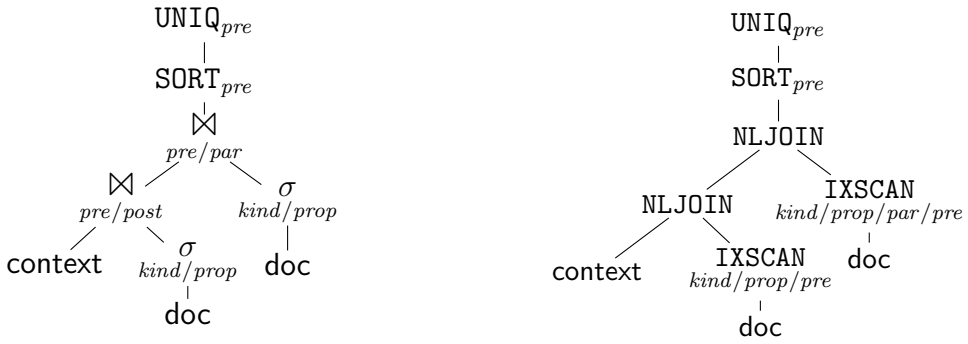
The XPath accelerator tree encoding leverages mature database technology into the domain of XML data and turns relational DBMSs into efficient tree processors. In this chapter, we will increase the tree awareness of relational database systems once more by the injection of a new join operator: *staircase join*. We will see that this single operator, when added to the DBMS kernel, encapsulates all tree knowledge that is inherent to the *pre/post* plane.

The chapter starts off with a review of some additional considerations about the XPath accelerator tree encoding. We will then introduce staircase join (Section 3.2) and discuss its implementation in disk-based or main memory-oriented DBMS kernels (Sections 3.3.1 and 3.3.2, respectively). Experiments confirm the effectiveness of staircase join in both setups. Section 3.4 discusses some more applications of the staircase join idea, before we conclude in Section 3.5 with a glimpse into the research vicinity of staircase join.

3.1 Re-Inspecting XPath Accelerator

In general, the translation of XPath expressions into *pre/post*-encoded data leads to relational query plans such as the one shown in Figure 3.1(a): a repeated self-join of the *pre/post* relation doc. Each XPath location step corresponds to a join operator in the relational plan, with the join condition determined by the predicates $axis(\alpha, v, v')$ and $test(\nu, v')$ (cf. Tables 2.1 and 2.2, respectively).

An RDBMS query optimizer will typically generate a query plan like the one



(a) Logical query plan.

(b) Associated execution plan.

Figure 3.1: XPath location paths translate into repeated self-joins of the document relation doc. (b) The resulting plan is typically executed in terms of *index nested loops joins* (query `context/descendant/following-sibling`).

shown in Figure 3.1(b) to answer queries for XPath location steps. XML documents are accessed in terms of an *index nested loops join*, where the indexed doc relation serves as the inner join relation. Depending on the join predicate and the availability of appropriate indexes, systems may even employ *index-only scans* and, hence, not access the base relation doc at all.

3.1.1 Node Distribution in the *pre/post* Plane

Proper index selection as well as efficient plan generation is typically driven by *statistics* on the underlying database tables. The non-uniform point distribution in the *pre/post* plane, however, makes such statistics a poor means to capture the tree-specific nature of the data. Off-the-shelf RDBMSs, *e.g.*, will not detect the presence of unpopulated regions in the *pre/post* plane. In effect, they will waste significant effort on scanning such unpopulated space. Making the system's optimizer aware of the underlying tree data could be a means to save a considerable amount of effort.

In case of the XPath `descendant` axis, such hints to the optimizer were expressible on the SQL level in terms of *shrink-wrapping* (cf. page 18) by adding the line

$$\texttt{AND}\ v_1.pre \leq c.post + height(t)\ \texttt{AND}\ v_1.post \geq c.pre - height(t)$$

to the SQL expression in Figure 2.3(b). As observed in Section 2.2.1, this leads to a significant improvement in query performance. In the following, we will see

that the awareness of similar tree properties can lead to even further speedups of XPath evaluation.

Expensive XPath Semantics

Another observation in Figure 3.1(a) is the need for expensive sorting and duplicate elimination (operators `SORT` and `UNIQ`, respectively) to implement the respective XPath semantics. In Chapter 2, we looked at location steps that originate from a single context node only. In general, however, an entire *set* of context nodes serves as the input to an XPath location step. In consequence, the joins in Figure 3.1(a) will typically produce duplicate nodes that the database system needs to expensively eliminate afterwards.

3.2 Staircase Join

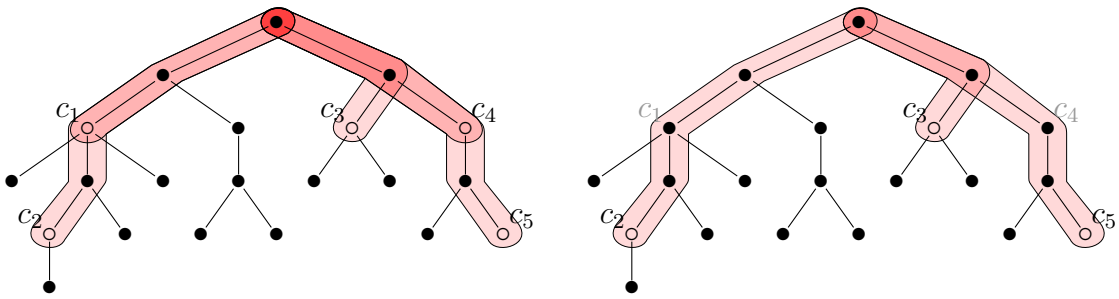
The processing of an entire set of context nodes is, in fact, the basis for the optimizations we present in this chapter. We will first look into such a situation in an XML document tree and see how it is reflected in the *pre/post* plane. This will reveal a number of interesting tree properties that we will exploit afterwards in terms of the three techniques *pruning*, *partitioning*, and *skipping*. The new join operator that we propose, *staircase join* (⛉), incorporates all three techniques and introduces *full tree awareness* if added to an RDBMS kernel. Though we will discuss its details based on the *pre/post* encoding, all staircase join ideas can be easily adapted to other encoding variants as well (the *pre/size/level* variant in particular).

3.2.1 Pruning

Effectively, the processing of a context set with an (index) nested loops join as we have observed it in actual DBMS implementations leads to the independent evaluation of XPath axis regions for each context node in turn. Figure 3.2(a) depicts the situation for the evaluation of an `ancestor-or-self` location step for a context set of five nodes.

The `ancestor-or-self` regions for these nodes significantly overlap, which we indicated by the intensity of the shadings in Figure 3.2(a). The darker a path's shade, the more often will the contained nodes appear in the join result, which ultimately leads to the requirement for the duplicate removal operator `UNIQ` in Figure 3.1(a).

Obviously, we can remove a node from the context set if its `ancestor-or-self` region is entirely included in the query region of some other context node(s) prior



(a) Original context set: $(c_1, c_2, c_3, c_4, c_5)$.

(b) Context set after pruning: (c_2, c_3, c_5).

Figure 3.2: (a) The `ancestor-or-self` regions for the context set $(c_1, \ldots, c_5)$ intersect and hence produce duplicates in the result. (b) The pruned context set (c_2, c_3, c_5) covers the same `ancestor-or-self` region with less duplicate nodes.

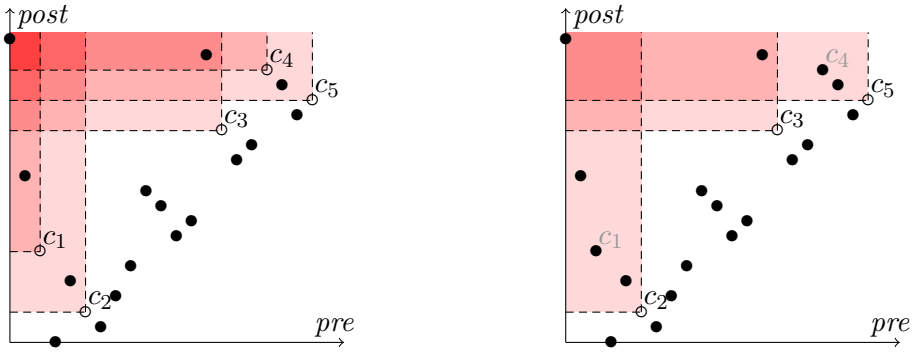
to join processing without any effect on the final query result. In our example in Figure 3.2, nodes c_1 and c_4 are instances of such nodes. As we see in Figure 3.2(b), the removal of such nodes, the so-called *pruning* step, minimizes the number of duplicates generated while, at the same time, lowering the join processing costs due to a reduced context size. Note, however, that the duplicate problem still persists: some tree nodes in Figure 3.2(b) (the root node in particular) are still reachable from more than one context node.

Context Pruning in the *pre/post* Plane

In case of the `ancestor-or-self` axis, a context node v is a candidate for pruning if it is located along a path from some other context node v' to the document root. Figure 3.3 illustrates this situation in the *pre/post* plane: each context node defines the boundary of its corresponding `ancestor-or-self` region. Regions may *include* one another or *partially overlap*.

The former case of *inclusion* is easily dealt with by the removal of covered nodes, and thus regions, from the context set. This is exactly how we removed the context nodes c_1 and c_4 before. Figure 3.3(b) illustrates this removal in the *pre/post* plane after pruning the context set of Figure 3.3(a).

Similar opportunities to simplify the context set arise for other XPath axes as well. Figure 3.4 illustrates the situation for the `descendant`, `following`, and `preceding` steps for the same context set as in Figure 3.3(a). Again, a number of context nodes and their query regions are entirely covered by the query regions of other nodes such that we can safely remove them from the context set, *i.e.*, we can remove (a) nodes c_2 and c_5 for the `descendant` axis, (b) nodes c_3, c_4, and c_5



(a) Original context set: $(c_1, c_2, c_3, c_4, c_5)$. (b) Context set after pruning: (c_2, c_3, c_5).

Figure 3.3: (a) Nodes may be pruned from the context set if their query regions include one another in the *pre*/*post* plane. (b) Pruning minimizes the region overlap and hence the production of duplicate result nodes (cf. Figure 3.2).

for the `following` axis, and (c) nodes c_1, c_2, and c_3 for the `preceding` axis.

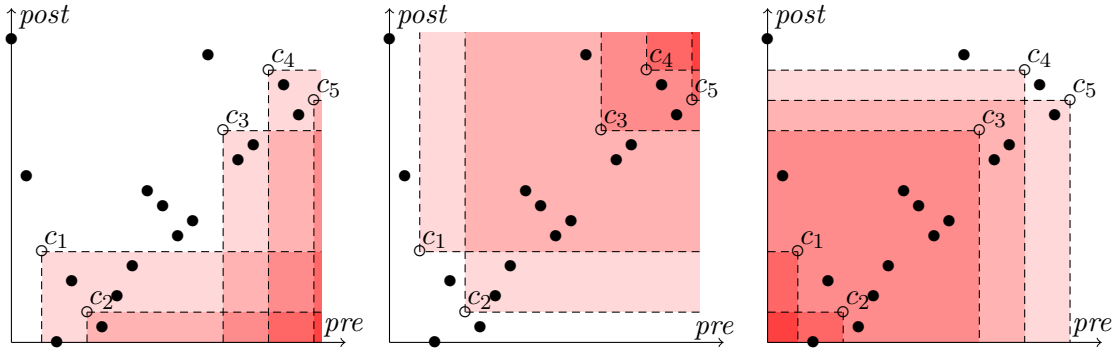
Pruning for Forward Axes

The task of identifying the inclusion of query regions is easily implemented for a *pre*/*post*-encoded context sequence. Algorithm 3.1 lines up the pruning procedure for the `descendant` axis. Routine `prunecontext_desc ()` assumes that the table `context` is sorted along the *pre* dimension and processes it in a single sequential pass. It collects nodes whose *post* value exceeds the maximum value seen so far and discards the others. Note that this implementation seamlessly integrates with the pipelined execution model of modern RDBMSs.

`prunecontext_desc (context :` TABLE $(pre, post)$`)`

```
1 BEGIN
2   result ← NEW TABLE (pre, post);
3   prev ← 0;
4   FOREACH c IN context DO
5     IF post(c) > prev THEN
6       APPEND c TO result;
7       prev ← post(c);
8 END
```

Algorithm 3.1: Context pruning for the `descendant` axis.



(a) `descendant` axis. (b) `following` axis. (c) `preceding` axis.

Figure 3.4: Overlapping query regions for context set $(c_1, \ldots, c_5)$. The darkly shaded areas are covered by the query windows of multiple context nodes and hence produce duplicate result nodes.

After pruning for the `descendant` axis, all remaining context nodes exclusively relate to each other in the `preceding`/`following` direction. As a result, they form a proper staircase as illustrated in Figure 3.5.

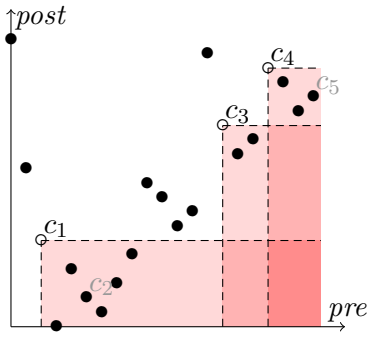
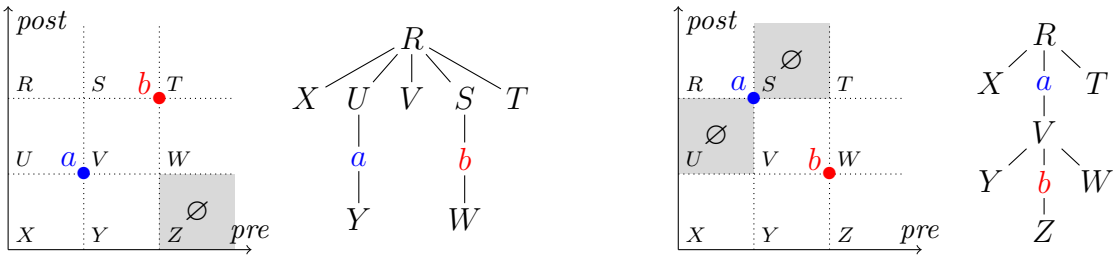


Figure 3.5: Pruned context set for Figure 3.4(a).

While Algorithm 3.1 is trivially adapted to implement context pruning for the `following` axis, the analogous handling of *reverse* axes (*e.g.*, `ancestor` or `preceding`) would require the context set to be read in reverse document (*i.e.*, *pre*-) order and, hence, block pipelined processing. It turns out, however, that these cases also become fully pipelineable if we consider further properties of the *pre*/*post* plane that are due to the tree origin of the nodes in the plane. These properties are what we will look into next.

3.2.2 Empty Regions in the *pre*/*post* Plane

Observe that the remaining region overlap in Figure 3.5 (darker shaded regions) is entirely empty. Apart from that, Figure 3.4 also exhibits some cases in which query regions cover unpopulated space in the *pre*/*post* plane. In Figure 3.4(b), context node c_1 does not contribute any new result nodes to the `following` region originating at node c_2. Respectively, node c_4 does not contribute to the `preceding` result of c_5 in Figure 3.4(c). Note that the situation persists even if we apply context pruning based on region inclusion.



(a) Nodes a and b relate on the `preceding/following` axis.

(b) Nodes a and b relate on the `ancestor/descendant` axis.

Figure 3.6: Empty regions in the *pre*/*post* plane.

The described cases are *not* a coincidence of the example chosen here, but a direct consequence of the fact that the *pre*/*post* plane represents the encoding of a *tree*. In fact, more nodes may be pruned from the context set if we introduce *tree awareness* into the pruning procedure.

Figure 3.6 gives a detailed insight into the tree-related properties of regions in the *pre*/*post* plane. For any two nodes a and b, there are only two cases in which they can relate to each other: (a) on the `preceding/following` axis or (b) on the `ancestor/descendant` axis. In either case, nodes a and b partition the entire *pre*/*post* plane into the nine regions R to Z illustrated in Figure 3.6.

If we assume both nodes to be in a `preceding/following` relationship (Figure 3.6(a)), the presence of a node v in region Z would imply that v is a descendant of both, nodes a and b. Obviously, such a situation cannot arise in a tree, and we can conclude that region Z must be entirely empty. (Note how this observation of an *empty region* explains the node distribution in the *pre*/*post* plane observed in Section 2.1.5.)

The case in which a and b relate in an `ancestor/descendant` direction is illustrated in Figure 3.6(b). Again, based on our knowledge about the plane's tree origin, we know that an ancestor of node b may never precede (region U) or follow (region S) a. Hence, U and S must be empty regions as well.

Pruning for Reverse Axes

The observations about empty regions form the basis for the pruning algorithm of the XPath `ancestor` step as shown in Algorithm 3.2. From Figure 3.6(a), we know that we can never encounter a context node whose `ancestor` region covers the `ancestor` region of any node *prev* already appended to the pruned context set, as such a node had to be situated in an empty region of type Z. This means

```
prunecontext_anc (context : TABLE (pre, post))
1 BEGIN
2   result ← NEW TABLE (pre, post);
3   prev ← FIRST NODE IN context;
4   FOREACH c IN context DO
5     IF post(c) > post(prev) THEN
6       APPEND prev TO result;
7     prev ← c;
8   APPEND prev TO result;
9 END
```

Algorithm 3.2: Context pruning for the `ancestor` axis.

that with `prunecontext_anc()` we have found a means to prune an `ancestor` context set in a fully pipelineable fashion (again we assume context to be sorted in document (*pre*-) order).

The observations about empty regions also allow for pruning opportunities in case of the `preceding` and `following` axes that are even more profound. Pruning based on region inclusion as discussed earlier leads to a context set where all nodes are related to each other along the `ancestor`/`descendant` axes, *i.e.*, they are all arranged in the same way as in Figure 3.6(b). For the `preceding` axis, this implies that we can prune all context nodes but the one with the maximum *pre* value, because region U is known to be empty. The same applies to the node with the minimum *post* value in case of the `following` axis where region S must be empty.

With only a single context node remaining, this degenerates the evaluation of the `preceding` and `following` axes into a single region scan, *regardless* of the original context size. We will, hence, focus on the `ancestor` and `descendant` axes in the following and come back to the `preceding` and `following` axes in Section 3.4.3.

3.2.3 Partitioning

Pruning redundant nodes from the context set may significantly reduce the amount of duplicates produced by an XPath location step. However, as we can see in Figures 3.2(b) and 3.3(b), a number of duplicates still remains due to intersecting `ancestor-or-self` paths originating at different context nodes.[1] We can avoid this if we *partition* the document tree using the remaining context nodes and

[1] In fact, the tree root is always returned once for every context node.

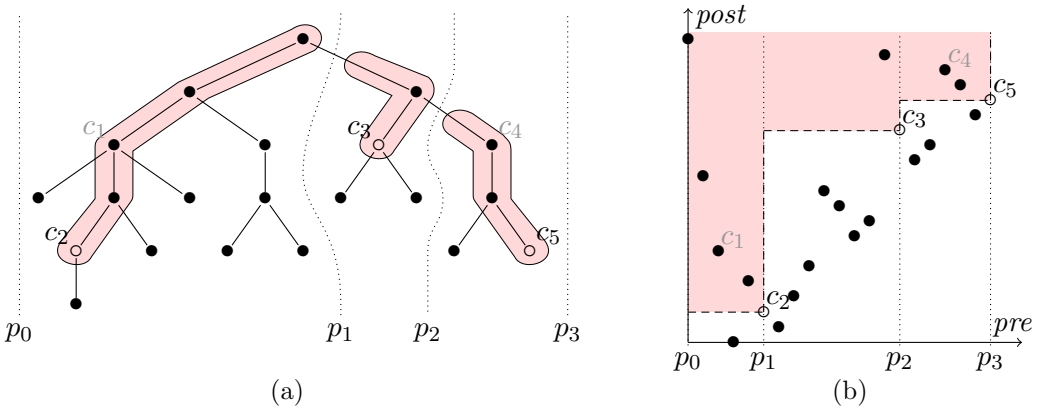


Figure 3.7: Document partitioning induced by context nodes c_2, c_3, and c_5.

ensure that we process each partition just once.

This approach is illustrated in Figure 3.7 for the case of the `ancestor-or-self` step that we have seen earlier. The context nodes' preorder ranks $pre(c_i)$ induce a partitioning of the *pre*/*post* plane that follows the staircase shape retrieved after pruning. Each of the partitions $[p_0, p_1]$, $(p_1, p_2]$, and $(p_2, p_3]$ defines a region within the plane that contains the axis step's sub-result for the context nodes c_2, c_3, and c_5 (respectively).

A natural way to process such a partitioned *pre*/*post* plane is listed in Algorithm 3.3. These basic staircase join algorithms sequentially scan the plane partition-wise from left to right, selecting those nodes in each partition that lie within the sub-result of its respective context node. Note that Algorithm 3.3 assumes both input relations, the context set context and the XPath accelerator document table doc, to be sorted in *pre*-order. Hence, accessing the tuple doc$[i]$ in procedure `scanpartition()` does not actually involve a distinct lookup for the tuple with the preorder rank i, but rather means using *the record at hand* during the sequential scan. Furthermore, as staircase join effectively traverses the tree in document order, it will automatically produce its result in document order, too.

The evaluation of a location step according to Algorithm 3.3 exhibits a number of interesting characteristics and—most importantly—upper bounds for the execution of each step:

(i) tables doc and context are both scanned *sequentially*,

(ii) both tables are read only *once* for an entire context set (hence, |doc| is an upper bound for the number of tuple accesses in relation doc),

(iii) staircase join *never* delivers duplicate nodes, and

```
staircasejoin_anc (doc : TABLE (pre, post), context : TABLE (pre, post)
1  BEGIN
2  |  result ← NEW TABLE (pre, post);
3  |  n ← FIRST NODE IN context;
4  |  d ← FIRST NODE IN doc;
5  |  scanpartition (pre(d), pre(n) − 1, post(n), >);
6  |  FOREACH SUCCESSIVE PAIR (n1, n2) IN context DO
7  |  |  scanpartition (pre(n1) + 1, pre(n2) − 1, post(n2), >);
8  |  RETURN result;
9  END
```

```
staircasejoin_desc (doc : TABLE (pre, post), context : TABLE (pre, post)
10 BEGIN
11 |  result ← NEW TABLE (pre, post);
12 |  FOREACH SUCCESSIVE PAIR (n1, n2) IN context DO
13 |  |  scanpartition (pre(n1) + 1, pre(n2) − 1, post(n1), <);
14 |  n ← LAST NODE IN context;
15 |  d ← LAST NODE IN doc;
16 |  scanpartition (pre(n) + 1, pre(d), post(n), <);
17 |  RETURN result;
18 END
```

```
scanpartition (pre1, pre2, post, θ)
19 BEGIN
20 |  FOR i FROM pre1 TO pre2 DO
21 |  |  IF post(doc[i]) θ post THEN
22 |  |  |  APPEND doc[i] TO result;
23 |  RETURN result;
24 END
```

Algorithm 3.3: Basic staircase join algorithms (`ancestor` and `descendant` axes). For each context node, procedure `scanpartition()` sequentially scans the associated *pre* interval in document order.

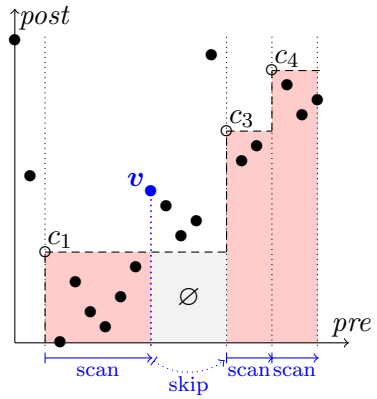


Figure 3.8: Skipping for the `descendant` axis.

```
scanpartition_desc (pre1, pre2, post)
1 BEGIN
2   FOR i FROM pre1 TO pre2 DO
3     IF post(doc[i]) < post THEN
4       APPEND doc[i] TO result;
5     ELSE
6       BREAK ;                    /* skip */
7   RETURN result;
8 END
```

Algorithm 3.4: Abort scanning early (line **6**) to implement skipping (`descendant` axis).

(iv) result nodes are returned in *document order*; no expensive post-processing is required to comply with the semantics of XPath.

It is important to note, though, that Algorithm 3.3 only works correctly on proper staircases, *i.e.*, on pruned context sets. Both processing stages, pruning as well as staircase join evaluation, read their input in a strictly sequential fashion. They may thus straightforwardly be integrated into a single algorithm. In [Grust03a], the authors describe a staircase join variant that implements such pruning *on the fly*.

3.2.4 A Further Increase of Tree Awareness: Skipping

If we know certain regions of the *pre*/*post* plane to be empty, we may as well try to *skip* them during our scan. Figure 3.8 illustrates this idea for the `descendant` axis.

To evaluate this axis, staircase join scans the *pre*/*post* table from left to right, starting at the first context node c_1. During the scan of the first document partition (associated with c_1), the first node we encounter *outside* of c_1's `descendant` boundary is node v. Since nodes c_1 and v are arranged in a `preceding`/`following` relationship as illustrated earlier in Figure 3.6(a), we can conclude that no further descendants of c_1 can be found between v and the next context node c_3 (indicated with ∅ in Figure 3.8). In such a situation, we may stop processing the current partition immediately and continue the evaluation at the next context node, as indicated by the skip in Figure 3.8.

It is easy to adapt the basic staircase join algorithm to this behavior by replacing its worker function `scanpartition()` by the implementation shown as

Algorithm 3.4. When a node outside the current *post* boundary is encountered, the algorithm immediately aborts the current partition scan (line **6**) and continues processing at the next document partition.

The effectiveness of skipping can be substantial. Each node we encounter while scanning the document partition of a context node c_i will either *(i)* belong to c_i's result set or *(ii)* be of kind v and, hence, lead to a skip. To produce the final result, we thus never touch more than $|\mathsf{context}| + |\mathsf{result}|$ nodes in the *pre/post* plane.

Most importantly, the effort to evaluate an XPath location step becomes *independent* of the size of the *pre/post* table (*i.e.*, of the XML document size). In contrast to that, the basic staircase join algorithm (Algorithm 3.3) would scan the entire document relation, starting at the context node with minimum preorder rank, and, hence, have an upper bound of $|\mathsf{doc}|$ tuple accesses. In [Grust03c], we found staircase join to skip about 92 % of these tuples.

Skipping for Other XPath Axes

For the `ancestor` and `following` axes, we can find similar opportunities to skip parts of the document relation:

Ancestor Axis. Whenever we encounter a node v outside the `ancestor` boundaries of some context node c, we know that all of its descendants will also not qualify as ancestor nodes of c. In that case, Equation 2.3 (that we used earlier to implement shrink-wrapping) comes in handy to provide an estimate on the size of the subtree below v. We are off by at most the height of the document tree if we skip to the node v' with $pre(v') = post(v)$ in that case.

Following Axis. Evaluating the `following` axis, we know that the nodes we encounter in *pre*-order immediately after the single remaining context node c will be c's descendants. Again, using Equation 2.3, we can skip those with an error of at most $height(t)$. Consequently, we will read at most $height(t)$ false hits to evaluate an XPath `following` step, resulting in an upper bound of $|\mathsf{result}| + height(t)$ node accesses.[2]

Preceding Axis. When scanning the *pre/post* region left of the single remaining context node c for a `preceding` step, we encounter all nodes that precede c, interspersed with the ancestors of c. There are at most $height(t)$ of the latter, such that even without skipping we have to read at most $|\mathsf{result}| + height(t)$ nodes to evaluate the `preceding` axis.

[2] Remember that $height(t) \ll |\mathsf{doc}|$.

3.3 Implementation Considerations

We have now identified a number of optimizations that a relational database engine could exploit for accelerated XPath step evaluation. Staircase join makes those optimizations available in the form of a new join operator which is added to the DBMS kernel. The ⌐ operator encapsulates all tree specifics present in the database's *pre/post* relation and uses this *tree awareness* to evaluate XPath location steps at a minimal cost. We claim that staircase join could easily be added to any relational database kernel, turning it into a high-performance tree processor.

Maybe the most suitable characterization of the staircase join algorithm is a merge join with a dynamic range predicate. ⌐ reads both of its inputs, a relation that represents the step's starting point and the document table doc, in a single sequential scan. Moreover, staircase join exhibits a number of valuable properties that allow for a seamless integration into existing database kernels:

(i) Staircase join is a join in the usual sense. As expected for other join implementations as well, ⌐ allows for effective plan rewrites such as, *e.g.*, selection pushdown.

(ii) Both input relations are consumed and processed in a stream-like fashion, making ⌐ not only interesting in terms of its cache-friendliness, but also suitable for pipelined execution.

Both aspects turn out to be equally important, whether staircase join is introduced into a traditional, disk-based RDBMS or into a modern database kernel tuned for CPU efficiency and execution in main memory. To prove the claim that ⌐ speeds up XPath execution in any RDBMS, we implemented the operator in the disk-based system *PostgreSQL* [PostgreSQL] as well as in the main memory DBMS *MonetDB* [Boncz02].

3.3.1 A Disk-Based ⌐ Implementation

To assess the applicability of staircase join to a disk-based database environment, the open-source system PostgreSQL provides quite an ideal playground. In the course of the work in [Mayer04a], we injected staircase join into PostgreSQL 7.3.3. The required changes to PostgreSQL were remarkably small. Only local changes to the database's execution engine (the addition of staircase join itself) and an adaption of its planner/optimizer (to make it actually use the new operator) led to significant performance advantages for queries on *pre/post*-encoded data. At the same time, the addition did not do any harm to non tree-related SQL queries.

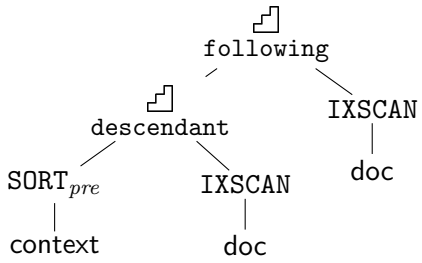


Figure 3.9: Staircase join-enhanced query plan for the path expression `context/descendant/following`. The ⍏ operator requires its input to be *pre*-sorted and guarantees the same for its output.

Scheduling for Staircase Join

To benefit from the physical staircase join operator, we modified PostgreSQL's planning/optimization phase to apply the new ⍏ operator whenever it detects an instance of an XPath location step. This decision is based on an examination of the join clauses in the query tree. We trigger the use of staircase join whenever the clauses meet the following four conditions:

(i) the join must be defined as two conditions on attributes named *pre* and *post*,

(ii) the respective comparison operators must specify a valid XPath axis (*e.g.*, $d.pre > c.pre$ and $d.post < c.post$ for the `descendant` axis),

(iii) both columns (*pre* and *post*) must be of data type `tree` in both join partners (This column type was newly introduced into PostgreSQL to identify columns that hold tree-structured data.), and

(iv) the system must find a suitable index on the inner join relation that supports the ⍏ operator, *i.e.*, a B-tree index on column *pre*.

The resulting plan (see Figure 3.9 for an instance) resembles the execution plan we saw earlier for an unmodified relational system (cf. Figure 3.1(b)), except that ⍏ is used to join the context set with the document relation for each XPath step. Observe how ⍏ reads its input in document order (hence, requires a `SORT` operator to access the context sequence). In return, it guarantees a duplicate-free result, that is returned in document order (hence, no additional sorting effort is required at the plan's root).

Indexed Document Access

As mentioned earlier, an unmodified PostgreSQL instance would execute path steps on *pre*/*post*-encoded data by applying its *index nested loops join* algorithm.

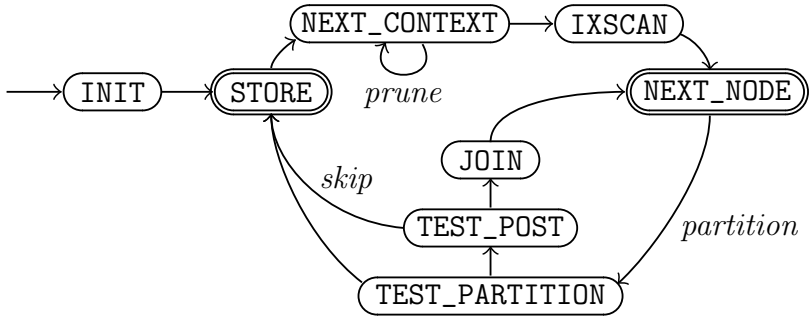


Figure 3.10: The finite state automaton that defines a fully pipelineable ⏎ implementation for the XPath `descendant` axis.

The *pre/post* relation doc is thereby accessed via a B-tree on column *pre*. PostgreSQL provides a special index variant for this purpose, the *inner join index*, that allows for efficient re-scans of the inner relation for each of the distinct tuples from the outer join relation.

Staircase join adopts this way of accessing the doc relation as shown in Figure 3.9 (operators `IXSCAN`). Our implementation initiates a partition scan at the lower partition boundary p_i by triggering an index (re-)scan of the document table using the condition doc.*pre* $> p_i$. The inner join index will deliver the matching tuples in ascending *pre*-order, starting directly at the first *pre*-value that satisfies this property. With pruning in place and for a *pre*-ordered context set, this leads to a progressive forward scan of doc.

A Pipelineable ⏎ Implementation

Typical for a disk-based database system such as PostgreSQL, it strives to avoid materialization and process data in a fully pipelined fashion. To appropriately support this execution paradigm, we carefully adapted the original staircase join algorithm (Algorithm 3.3) such that it will only request the next input tuple from either subplan if it is immediately required to produce the next output tuple.

The clearly distinguished execution steps predefined by pipelining and the three staircase join-specific techniques (pruning, partitioning, and skipping) suggested the use of a *finite state automaton* to implement the staircase join execution module. Each of the four major axes was assigned its own automaton.

Figure 3.10 illustrates the automaton that implements staircase join for the XQuery `descendant` axis. After the first context tuple v_1 has been read from the outer subplan (state `INIT`), the automaton stores it as the lower boundary for the first partition. State `NEXT_CONTEXT` then requests tuples from the outer subplan (the context set), until a context node v_2 is found with a higher *post* value than

v_1. This implements context set pruning (cf. `prunecontext_desc ()`, page 43).

As soon as the first scan partition, $(v_1, v_2]$, is set, ⏍ starts reading the document nodes within that partition. As mentioned before, we assume the `doc` relation to be accessed via an inner join index and, hence, initiate an index re-scan (state `IXSCAN`), starting right after $pre(v_1)$. Tuples are retrieved one by one from the inner join index in state `NEXT_NODE`. For each of them we ensure that it is still within the boundaries of the current scan partition (state `TEST_PARTITION`) and that it satisfies the *post* condition (state `TEST_POST`). State `JOIN` then builds and returns the next result tuple. As soon as we encounter a node outside the current scan partition (state `TEST_PARTITION`), we switch to state `STORE` and, hence, to the next partition.

As we have seen, the crucial ingredient of efficient step evaluation with staircase join is the opportunity to *skip* over parts of the *pre*/*post* table that are known not to contribute to the result. In case of the `descendant` axis, this opportunity arises whenever `scanpartition_desc ()` (Algorithm 3.4) encounters a node outside the current *post* range. In our pipelineable implementation, the state automaton immediately switches to the `STORE` state whenever it finds a node that does not meet the current *post* condition (state `TEST_POST`) and skips forward to the next context node.

The automaton reaches a final state if either the outer or the inner subplan runs out of tuples.

Quantitative Assessment

We used our staircase join-enhanced PostgreSQL instance to re-run queries Q_1–Q_4 from Chapter 2 on the identical hardware. For ease of comparison, we also recite the execution times we observed in Chapter 2. All execution times are illustrated in Figures 3.11 (Query Q_1, `descendant` axis), 3.12 (Query Q_2, `ancestor` axis), and 3.13 (Queries Q_3/Q_4, `preceding`/`following` axes).

The four queries each stress one specific axis of the XPath language. On our original PostgreSQL platform, we observed a quadratic scaling to document sizes for each of them on documents generated with `xmlgen`. Only explicit modifications to the original query (in the case of Q_2) or to the resulting SQL code (Query Q_1) were able to reduce this complexity to a linear behavior. In contrast, none of these modifications are required if the database has been extended by staircase join, which incorporates any tree-specifics present in the *pre*/*post* plane. Our enhanced PostgreSQL installation beats the original instance in all of our test queries (even the modified ones). Staircase join achieved linear scalability within the entire size range.

The staircase join-extended PostgreSQL system shows its advantage most prominently during the execution of the `preceding` and `following` axes (Queries Q_3

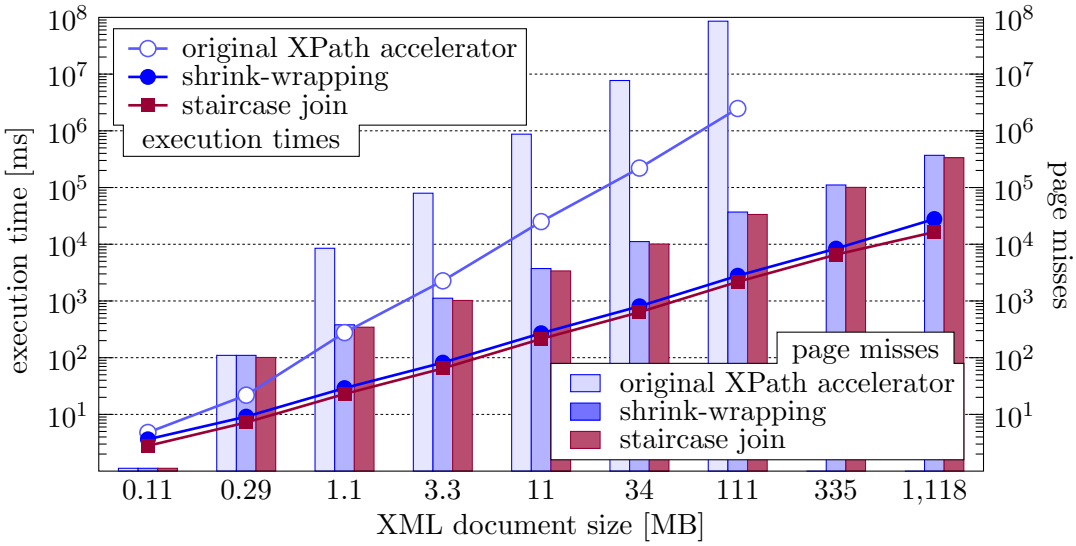


Figure 3.11: Staircase join performance for Query Q_1. Staircase join achieves linear performance in the query's result size. The introduction of the new join operator no longer requires explicit query adaptions on the SQL level.

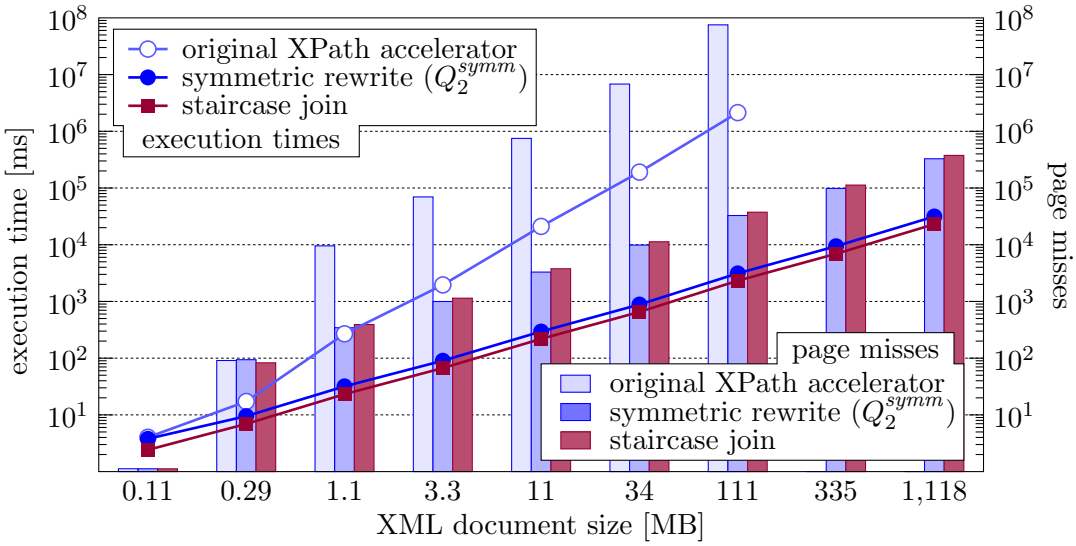
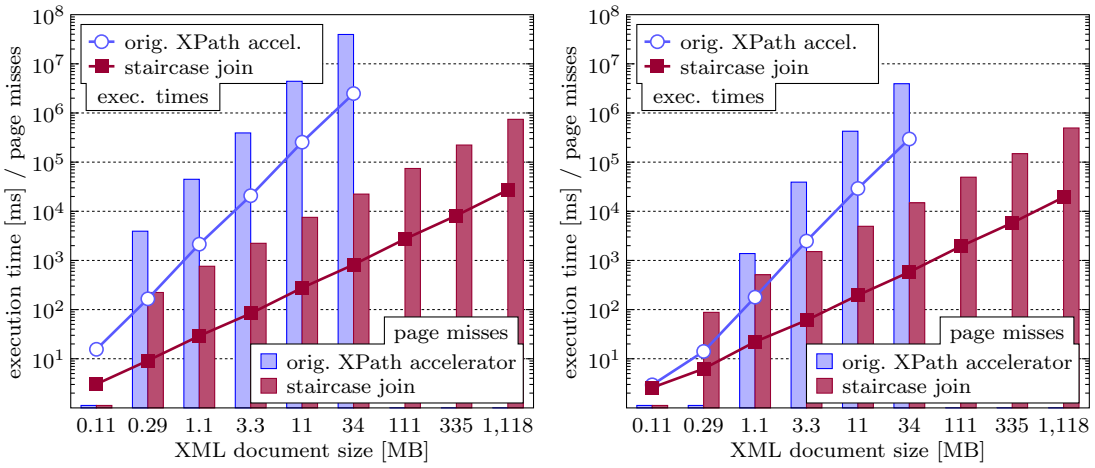


Figure 3.12: Staircase join implementation for the `ancestor` axis (Q_2). The enhanced PostgreSQL instance renders any rewrites to the XPath expression itself obsolete.



(a) Query Q_3 (`preceding` axis). (b) Query Q_4 (`following` axis).

Figure 3.13: Staircase join on PostgreSQL for the `preceding` and `following` axes.

and Q_4). Here, the off-the-shelf system not only suffers from the significant *search* effort; *sorting* the query result and eliminating duplicate hits add another considerable cost. In fact, the original system was not able to handle the two queries for XML instances above 34 MB at all (Figures 3.13(a) and 3.13(b)).

For further details and experiments on the injection of staircase join into PostgreSQL, we refer the interested reader to [Mayer04a]. For now, we will put the disk-based execution paradigm aside and look into the domain of main memory-oriented database kernels.

3.3.2 Main Memory-Related Adaptions

The classical design of disk-based database systems considers access to secondary storage as the dominating cost factor. This model, however, is increasingly being invalidated by ongoing advances in hardware technology that put main memory access cost into the spotlight. *Main memory database systems* (MMDBMS) acknowledge this fact with algorithms optimized for execution on modern computing hardware. The careful consideration of CPU and cache characteristics is crucial to this new database paradigm, into which staircase join turns out to fit perfectly.

The *MonetDB* system [Boncz02] constitutes an advanced implementation of the main memory-oriented execution paradigm and is available in open source.[3] MonetDB's data model exclusively uses binary tables ("binary association tables",

[3] `http://monetdb.cwi.nl/`

BATs) and thus blends perfectly with the two-column schema of the XPath accelerator document encoding. The MonetDB kernel forms the back-end of the XQuery system *MonetDB/XQuery*. This section focuses on the staircase join implementation included therein.

BATs provide a number of interesting features with respect to the implementation of the XPath accelerator document encoding, in particular the column type `void` (*virtual oid*) mentioned earlier in Section 2.3.3. Columns of this type hold continuous sequences of integers $o, o+1, o+2, \ldots$, of which MonetDB only stores the offset o. It is not the reduced storage requirement that is of special interest here, but the possibility to execute many operations as *positional lookups*. This functionality provides a highly efficient means to implement staircase join's skipping idea.

The adaptions to staircase join that we are about to describe are motivated by the calculations we published in [Grust03c]. Though their concrete numbers are specific to a certain hardware system, these calculations exemplify hardware-independent characteristics of staircase join and may easily be reproduced for different systems. The figures in the following assume a Dual-Pentium 4 Xeon machine, clocked at 2.2 GHz. The system is equipped with 2 GB of main memory and a two-level cache (levels L_1/L_2) of sizes 8 KB/512 KB. L_1/L_2 have cache line sizes of 32 bytes/128 bytes with miss latencies of 28 cy/387 cy = 12.7 ns/176 ns (measured with the calibrator tool described in [Manegold02]).

CPU Bottleneck Elimination

In MonetDB, staircase join essentially nests two loops that scan the `context` and `doc` BATs, respectively. The major fraction of work is done by the inner loop, *i.e.*, by procedure `scanpartition_desc()` (see Algorithm 3.4), which scans the document relation `doc`. To optimize the CPU characteristics of staircase join, we will first concentrate on how to make this scanning process most efficient.

As suggested before, column *pre* forms a continuous sequence of integers, which perfectly matches MonetDB's space-efficient column type `void`. This leaves us with the storage of *post* values, which we implemented as 4-byte integers. No more overhead is required to store a tuple $\langle pre, post\rangle$ in MonetDB; hence, an L_2 cache line will contain $^{128}/_{4} = 32$ nodes. The instruction latencies of single assembly instructions [Int05] provide an estimate for the number of CPU cycles spent on each iteration in `scanpartition_desc()`. On our test machine, they amount to 17 cy per iteration. For one cache line, this is 17 cy × 32 = 544 cy which exceeds the L_2 miss latency of 387 cy; `scanpartition_desc()` is obviously CPU-bound (rather than cache-bound).

Algorithm 3.4 basically performs two distinct tasks for each document node: *(i)* test the *post* condition in line **3** and *(ii)* append qualifying nodes to the output

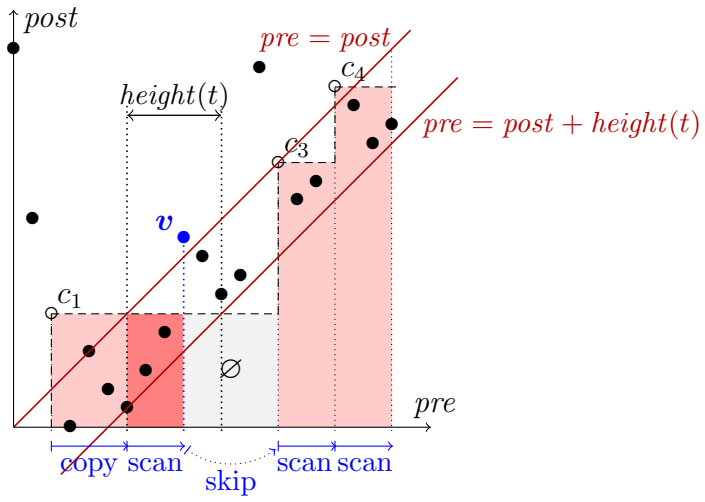


Figure 3.14: Estimation-based skipping: Lower and upper bounds for the number of descendants of each document node form two diagonals in the *pre*/*post* plane. The lower bound $pre = post$ defines the range of the copy phase in Algorithm 3.5.

BAT (line **4**). We obviously cannot avoid the latter step, as it implements the construction of the operator's result. However, we can save a large amount of work with respect to the former task if we, again, take tree-specific properties of values $pre(v)$ and $post(v)$ into account.

Based on Equation 2.3, we can find a lower bound for *pre* values, such that nodes below this *pre* boundary cannot possibly violate the join's *post* condition (This is dual to the shrink-wrapping conditions, *i.e.*, Equations 2.5.):[4]

$$pre(v') \leq post(v) \;\Rightarrow\; post(v') < post(v) \;\; . \tag{3.1}$$

According to Equation 2.3, this estimation of the number of v's descendant nodes is off by $level(v)$, *i.e.*, at most $height(t)$. In combination with Equation 2.5a, it defines lower and upper bounds for the size of each node's `descendant` region. In Figure 3.14, these bounds are illustrated by the two diagonals in the *pre*/*post* plane.

Nodes within the lower boundary may safely be copied directly into the result BAT, without any inspection of their *post* values. Our CPU-optimized implementation of `scanpartition_desc()` (Algorithm 3.5), hence, divides the processing of the `doc` BAT into two distinct phases:

(i) The *copy phase* directly copies the first $post(c) - pre(c)$ nodes that immediately follow the context node c in the *pre*/*post* table into the join result.

[4] We assume $pre(v') > pre(v)$ here, as this is the situation in `scanpartition_desc()`.

```
scanpartition_desc (pre1, pre2, post)
1  BEGIN
2    estimate ← max (pre1, post);
     /* copy phase                                              */
3    FOR i FROM pre1 TO estimate DO
4      APPEND doc[i] TO result;
     /* scan phase                                              */
5    FOR i FROM estimate + 1 TO pre2 DO
6      IF post(doc[i]) < post THEN
7        APPEND doc[i] TO result;
8      ELSE
9        BREAK ;                                    /* skip */
10   RETURN result;
11 END
```

Algorithm 3.5: A CPU optimized implementation of `scanpartition_desc()`.

According to Equation 3.1, no postorder ranks have to be checked for these nodes. The code for this phase forms an extremely tight copy loop.

(ii) The remaining document nodes within the partition are processed in the algorithm's *scan phase*. Essentially, this phase is identical to the original loop and includes a test for each node's postorder rank. As before, we abort scanning the current partition, as soon as we find the first node in c's `following` region and skip forward to the next context node.

As mentioned earlier, the estimation error with respect to the number of descendant nodes maximally amounts to the tree's height, $height(t)$. This is insignificant in comparison to the overall document size, so that the copy phase will represent the bulk of the algorithm's work. A single node copy iteration takes about 5 CPU cycles. On our system, processing one L_2 cache line now takes $5\,\text{cy} \times 32 = 160\,\text{cy}$ which lies clearly below L_2 miss latency. Procedure `scanpartition_desc()` is now cache-bound, rather than CPU-bound.

The *branch prediction friendliness* of both processing phases adds another share to the CPU efficiency of `scanpartition_desc()`. Both loops are terminated by a fixed end condition and the postorder condition always enters the same (THEN) branch, except for the last iteration. This branching behavior is perfectly predictable, such that the significant penalties for instruction retirement are avoided [Ross02].

Enhancements to Staircase Join's Cache Friendliness

Thanks to its sequential and strictly forward processing, staircase join shows a highly cache-friendly data access pattern, which we already found beneficial for disk-based database systems. This feature applies even more to the main memory-optimized domain.

On the hardware level, sequential tuple access leads to a full usage of CPU cache lines. The resulting sequential memory bandwidth of a machine with two cache levels has been described by [Manegold02]. For our example system, we get [Grust03c]:

$$\frac{LS_{L_2}}{L_{L_2} + \frac{LS_{L_2}}{LS_{L_1}} \times L_{L_1}} = \frac{128\,\text{byte}}{176\,\text{ns} + \frac{128\,\text{byte}}{32\,\text{byte}} \times 12.7\,\text{ns}} = 551\,{}^{\text{MB}}/_{\text{s}}$$

(with LS_C the cache line size of cache C and L_C the associated miss latency).

This bandwidth is shared by the parallel read and write operations in algorithm `scanpartition_desc()` (both employ strictly sequential memory access). During the execution of the procedure for the query `/descendant::node()` (which almost entirely consists of a copy phase), we observed a bandwidth that was significantly higher:

$$\begin{aligned}
&\frac{\textit{bytes read} + \textit{bytes written}}{\textit{execution time}} \\
&= \frac{(|\mathsf{doc}| + \mathsf{context}\ \textit{nodes scanned} + \textit{result size}) \times 4\,\text{byte}}{\textit{execution time}} \\
&= \frac{(50,844,982 + 1 + 47,015,212) \times 4\,\text{byte}}{519\,\text{ms}} \\
&= 719\,{}^{\text{MB}}/_{\text{s}}\ .
\end{aligned}$$

This excess sequential memory bandwidth is due to *hardware prefetching* provided by modern processors, such as the Pentium 4 system under investigation. After a short startup penalty—after which it has recognized the sequential access pattern—the CPU will automatically read *two* L_2 cache lines (*i.e.*, 256 byte) ahead.

An additional read ahead may be obtained by employing *software prefetching*. Explicit prefetch instructions (`prefetchnta`) advise the CPU to prefetch further cache lines ahead. Our system has an overall miss latency ($L_{L_1} + L_{L_2}$) of 28 cy + 387 cy = 415 cy, while Algorithm 3.5 requires only 2×160 cy = 320 cy to process the two cache lines loaded by the hardware prefetcher. Hence, we use `prefetchnta` to advise the CPU to read *three* cache lines ahead. In combination with additional code tuning (loop unrolling), this boosted the observed bandwidth up to 805 ${}^{\text{MB}}/_{\text{s}}$.

Performance Assessment

To assess the effect of introducing tree awareness into a main memory DBMS, we implemented staircase join as an extension module to the MonetDB system. As usual, documents from the XMark benchmark set [Schmidt02] served as a synthetic, yet realistic data set to examine the benefits of staircase join.

The tree-aware optimizations introduced with staircase join—pruning, partitioning, and skipping—are most effective for the XPath `descendant` and `ancestor` axes.[5] Figures 3.15 and 3.16 illustrate the performance of staircase join for the following queries (respectively):

- Q_{desc}: `/descendant::profile/descendant::education` and
- Q_{anc}: `/descendant::increase/ancestor::bidder`.

Again, the sole purpose of the initial step is to provide a sufficiently large context set for the second step, which is of actual interest. Both queries have been evaluated in [Grust03c] on the hardware described above. An IBM DB2 installation on the same system serves as a baseline for a commodity RDBMS (without the availability of staircase join).

Both queries were answered by our staircase join implementation in interactive time. Result sizes for Q_{desc} and Q_{anc} show linear scaling to document size and, hence, so do the execution times of staircase join.

Staircase Join and the `descendant` Axis. In case of Q_{desc}, Figure 3.15 shows that DB2 keeps up with staircase join fairly well for the full range of document sizes. This is mainly an effect of the shrink-wrapping conditions (included in Q_{desc}) that serve as a very accurate estimate to "simulate" staircase join's skipping technique.

Furthermore, a look into the execution plans employed by DB2 reveals another instance of its elegant means to evaluate XPath name tests: pushing them through the join operators makes them efficiently supported by suitable B-trees with a primary ordering on tag names (cf. page 31). As mentioned earlier, staircase join behaves much like a join in the classical sense to the query optimizer and name test pushdowns are equally applicable here. To demonstrate their effect, Figure 3.15 shows the execution times for both: staircase join with name tests pushed through ("early name test") and staircase join with name tests evaluated after join processing. Note, however, that the effectiveness of this plan rewrite depends on the selectivity of the particular name test. Ideally, the choice for either plan should be driven by a suitable *cost model*.

[5] With pruning in place, axes `preceding` and `following` degenerate into a single range scan. We will demonstrate this effect later in Section 3.4.3.

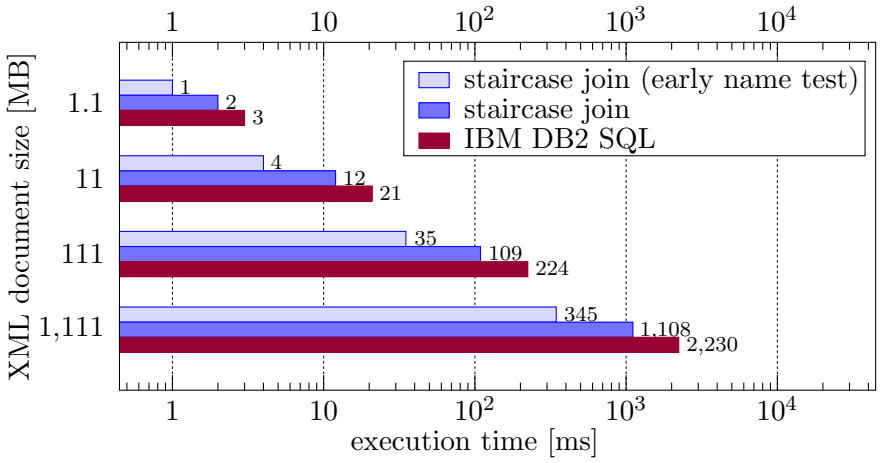


Figure 3.15: CPU-optimized staircase join performance observed for the evaluation of `/descendant::profile/descendant::education` (Q_{desc}). Name test pushdown saves another factor of three (measurements obtained in [Grust03c]).

Partitioning Boosts the `ancestor` Axis. The same arguments hold for the evaluation of an `ancestor` step, illustrated by Query Q_{anc} in Figure 3.16. Again, we can significantly benefit from early name test evaluation.

Though the actual SQL input we shipped to DB2 describes a symmetric equivalent to the Query Q_{anc} listed above (cf. Section 2.1.6) and despite the database's efficient application of *single record* index accesses (cf. Section 2.2.1), the execution times observed for DB2 significantly suffer from the involved *sort overhead*. The use of DB2's index nested loops join algorithms now falls behind the order-preserving staircase join operator by a factor of 25.

3.4 Tree Awareness Beyond Staircase Join

While staircase join elegantly integrates a high degree of tree awareness into a single database operator, it does not yet mark the end of tree-specific optimizations to relational database systems.

3.4.1 Loop-Lifting Staircase Join

It is in the very nature of staircase join to *prune* its input for redundant context nodes or overlap of query regions. While this is an ideal strategy for expressions along the lines of XPath 1.0, this optimization may hinder the implementation of language features that go beyond those simple path expressions, in particular XQuery's `FLWOR` expressions. While the implementation of this iteration construct

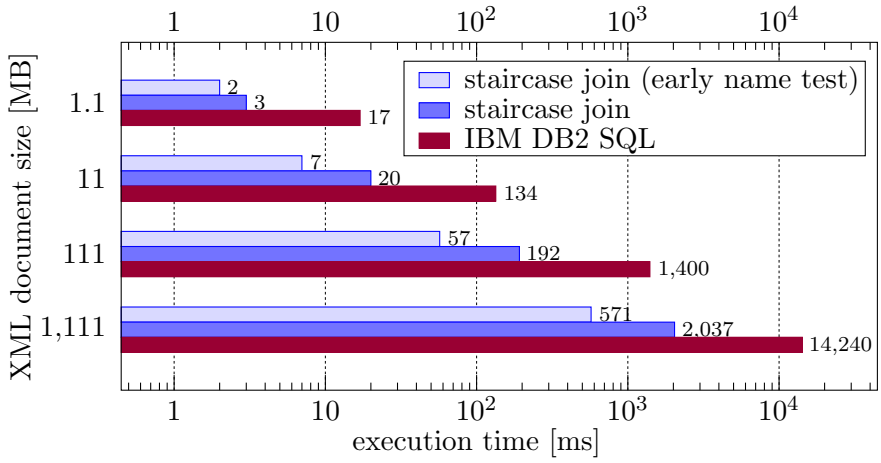


Figure 3.16: Path `/descendant::increase/ancestor::bidder`. The avoidance of duplicates is particularly effective for the `ancestor` axis. Staircase join's advantage over DB2 is even more significant here (measurements obtained in [Grust03c]).

itself will be the subject of the next chapter, we will look into the impacts on path evaluation right now.

Semantically, the use of a path expression within a `FLWOR` statement's `return` clause demands the repeated evaluation of the same sequence of path steps for multiple context sets. To exemplify, the expression

```
for $v in (/child::a, /child::a/child::b) return
  $v/descendant::node()
```

assembles the results of `/child::a` and `/child::a/child::b` into a single node sequence, and then, for each $v within the sequence in turn, queries $v's descendants. All of them are collected, in order, into the flat result sequence.

Note that this is *not* the same as evaluating

```
(/child::a, /child::a/child::b)/descendant::node() .
```

In the latter query, all nodes returned by `/child::a/child::b` could obviously be removed from the context set in order to avoid duplicates during the evaluation of the final `descendant` step. The former `FLWOR` query, in contrast, makes these duplicates a semantically required part of the result. Hence, the direct application of staircase join to the binding sequence `(/child::a, /child::a/child::b)` would violate the semantics of `FLWOR` expressions. A repeated execution of staircase join is hardly an alternative: for an iteration over n bindings, this could lead to n repeated document scans—an unacceptable effort for larger n.

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

In [Boncz05b], we introduced a *loop-lifted* variant of staircase join that allows for the evaluation of an XPath location step for *multiple* context sequences in a *single* document scan. A relational representation of the input accepted by loop-lifted staircase join is shown as the table on the left. For each of the n iterations (encoded in column *iter*), the table lists the preorder ranks $\gamma_{i,1}, \ldots, \gamma_{i,s_i}$ of all nodes that contribute to the context sequence in iteration $i \in \{1, \ldots, n\}$ (s_i denotes the length of the ith context sequence). In this representation, loop-lifted staircase join restricts pruning to nodes within each *iter* group.

Partitioning in the Loop-Lifted Case

It is a consequence of the *partitioning* technique in the basic staircase join algorithm that only a single context node is *active* (and hence "producing" result nodes) at a time while scanning the *pre*/*post* plane. Within an XQuery `FLWOR` clause, however, the same node(s) might occur more than once in the result, each occurrence belonging to a different iteration.

The key to partitioning for the *loop-lifted* staircase join variant is the maintenance of a *stack* of context nodes, annotated with the iterations they occur in. While scanning/skipping over the document, partition boundaries now determine when to push a context node onto the stack and when to pop it off again. When loop-lifted staircase join encounters a result node, it will analyze the stack and produce result tuples for all active context nodes/iterations.

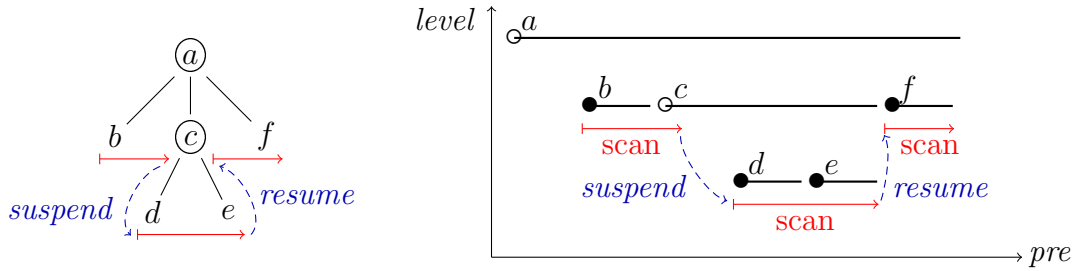
Main Characteristics of Loop-Lifted Staircase Join

We refer the reader to [Boncz05b] for details on the loop-lifted staircase join implementation but will summarize its main characteristics here:

(i) Loop-lifted staircase join processes an arbitrary number of context sequences in a single sequential document read. Skipping further reduces the number of nodes examined.

(ii) The algorithm's output is an $\langle iter, pre \rangle$ schema, ordered on $\langle pre, iter \rangle$. With only minimal effort, the result may alternatively be produced in $\langle iter, pre \rangle$ order.

The implementation of loop-lifted staircase join in the MonetDB kernel is part of the MonetDB/XQuery system, the open-source XQuery processor that accompanies this thesis.



(a) Sample tree. (b) Corresponding *pre*/*level* space.

Figure 3.17: Order-preserving `child` implementation for arbitrary context sets. If, while scanning the children of context node a, we pass a second context node c, we suspend the scan process and resume after collecting the children of c.

3.4.2 Support for Non-Recursive Axes

So far, the considerations about tree properties for efficient XPath evaluation were focused on the support for *recursive* axes. Similar observations, however, apply to the non-recursive case as well.

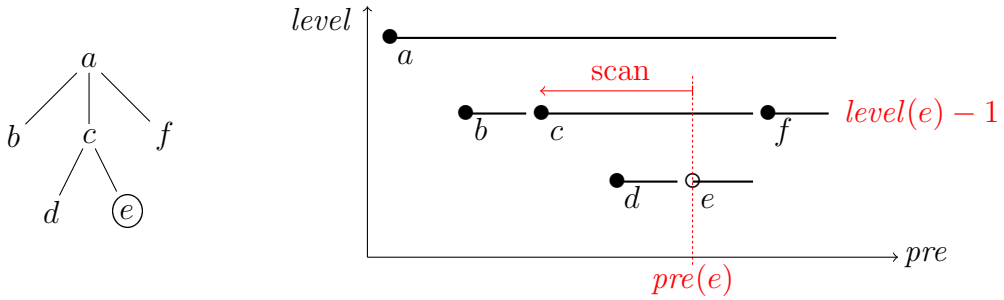
Evaluating the `child` Axis for Entire Context Sets

Querying along XPath's `child` axis was found to be highly efficient for single context nodes if the execution is backed by suitable B-tree indexes. On *pre*/*size*/*level*-encoded data, a range scan along a concatenated $\langle level, pre \rangle$ index for example, directly retrieves all children of a given context node in document order, without encountering any false hits (cf. page 30).

However, if applied to all items in a context sequence one by one, the result of such an approach will typically violate the XML document order. If we evaluate the query (a, c)`/child::*` in the sample tree in Figure 3.17(a) for both context nodes a and c in separation, the concatenated result (b, c, f, d, e) does not satisfy XPath's order semantics. In Figure 2.14, this led to the introduction of an explicit `SORT` operator into the query plan employed by DB2.[6]

Figure 3.17(b) illustrates an alternative, order-preserving technique to evaluate the `child` axis. The idea is to scan the `child` region of the first context node (a in Figure 3.17) using a $\langle level, pre \rangle$ index as usual. However, as soon as we pass the *pre* value of the next context node c in the set, we *suspend* the scan process

[6] The `child` axis does preserve uniqueness, though. If evaluated on a duplicate-free context set, the result of a `child` step will always be duplicate-free as well.



(a) Sample tree. (b) Corresponding *pre*/*level* space.

Figure 3.18: Tree-aware `parent` axis evaluation. On a concatenated $\langle level, pre\rangle$ index, a reverse scan starting at $\langle level(e) - 1, pre(e)\rangle$ will yield e's parent as its first hit.

and initiate a new scan for all children of c. As soon as we finish scanning for c's children, we *resume* the scan for the children of a.

This approach is inspired by the algorithm proposed in [Rode03] and guarantees a properly sorted step result for arbitrary context sets. Its execution requires exactly |result| tuple accesses. As the `child` axis naturally does not produce duplicate result nodes (hence, does not require pruning), this technique of retrieving child nodes straightforwardly extends to a loop-lifted variant.

A `parent` Step With Tree Awareness

Unfortunately, the idea of scanning a concatenated $\langle level, pre\rangle$ index to evaluate the `child` axis on range-encoded data cannot be directly transferred to the evaluation of XPath's `parent` axis. For that axis, the respective region conditions describe a constraint on *two* columns ($pre(v')$ and $size(v')$) that can no longer be mapped to a single B-tree scan:

$$v' \in v/\texttt{parent}$$
$$\Leftrightarrow$$
$$pre(v') < pre(v) \leq pre(v') + size(v') \ \wedge \ level(v') = level(v) - 1 \ .$$

However, considering tree-specific properties of the *pre*/*level* space, we may evaluate `parent` in a single index lookup nevertheless. For a given context node e (see Figure 3.18), a tree-aware `parent` implementation may trigger a reverse index scan[7] on a concatenated $\langle level, pre\rangle$ index, starting at $\langle level(e) - 1, pre(e)\rangle$. Such

[7] The functionality to scan indexes in a reverse fashion may presuppose an explicit declaration during index creation. On DB2, *e.g.*, reverse scans are enabled via the `CREATE INDEX ... ALLOW REVERSE SCANS` command.

a scan will always encounter e's parent node as its first hit (if any).

While this approach to the evaluation of the `parent` axis perfectly blends with the execution model of existing databases (DB2, *e.g.*, allows index scans to be executed as *single record* scans; cf. page 27), the choice of such an execution plan requires explicit knowledge about the data's tree origin. The technique sketched in Figure 3.18 is thus hardly expressible in SQL.

Additional advantages for the evaluation of the `parent` axis may be gained if we consider the step's behavior with respect to order. On a sorted context set, the one-by-one evaluation of the `parent` axis guarantees document order as well [Fernández04]. Hence, no explicit sorting is required to process the step's result.

3.4.3 Staircase Join Without Staircase Join

Staircase join is a single query operator that encapsulates full tree awareness and may turn existing relational database systems into efficient tree processors. Though the necessary changes remain local to the system's query engine, the addition of staircase join still requires an invasion into the respective database kernel. In some cases, however, this is not desirable or a system may not allow the addition of kernel operators at all.

Yet, many of the observations that led us to staircase join may still be made available to such "black box" systems. We have already discovered two such optimizations with a considerable impact in Chapter 2: shrink-wrapping and symmetric rewrites. Shrink-wrapping was identified as a weaker means to describe staircase join's *skipping* technique and makes this technique applicable to the symmetric rewrites. Both techniques were easily expressible in SQL, hence, accessible to commercial RDBMSs such as DB2.

In fact, the idea of *pruning* can have an equally significant impact on such systems. Pruning proved particularly effective for the `preceding` and `following` axes, where context sets of arbitrary size could be reduced to a single context node (cf. page 46). Both cases are easily expressible in SQL. *E.g.*, for the path e/`preceding::`ν, we get

$$
q(e/\texttt{preceding::}\nu) \equiv
\begin{array}{l}
\texttt{SELECT DISTINCT } v'.* \\
\quad \texttt{FROM doc AS } v, \texttt{doc AS } v' \\
\texttt{WHERE } v.pre \texttt{ = (SELECT MAX(}pre\texttt{) FROM } q(e)\texttt{)} \\
\quad \texttt{AND } v'.pre < v.pre \texttt{ AND } v'.post < v.post \\
\quad \texttt{AND } test(\nu, v') \\
\texttt{ORDER BY } v'.pre \ .
\end{array}
\tag{3.2}
$$

We ran the "pruned" SQL code for Queries Q_3 and Q_4 from Chapter 2 on our DB2 installation. The measured execution times are lined up in Figures 3.19

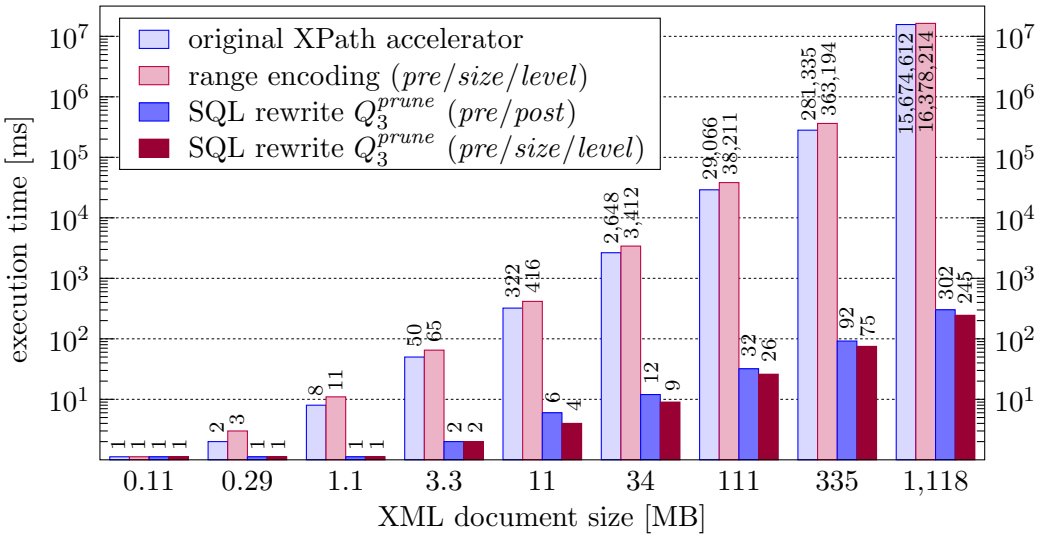


Figure 3.19: Rephrasing Query Q_3 on the SQL level leverages XPath performance by orders of magnitude (Q_3: `/descendant::current/preceding::initial`).

and 3.20. For reasons of comparison, both figures also include the execution times observed for the unmodified queries as presented in Section 2.2.1.

On both document encodings (XPath accelerator as well as range encoding), rephrasing the SQL code speeds up query evaluation by several orders of magnitude. Note that this is not only the effect of a reduced search effort due to the smaller context set. Moreover, the new queries produce significantly less duplicates and, hence, avoid a large amount of expensive result sorting.

3.4.4 Tree Awareness in Other Domains

The inspection of tree specifics hidden in encoded XML data is not restricted to the XPath evaluation domain. Another possibly performance-critical application is the efficient *validation* of XML tree fragments. This functionality, an integral part of the W3C XQuery specification, describes the verification of structural constraints on XML tree fragments, accompanied by an *annotation* of *type information* to the validated nodes.

Grust and Klinger [Grust04b] describe a validation procedure that generates type annotations for encoded tree data in a single sequential document read. Their approach may benefit from type annotations already attached to parts of the overall tree (such situations arise, *e.g.*, after the construction of tree fragments in XQuery's `strict` construction mode). In that case, detailed knowledge about the underlying encoding helps the algorithm to *skip* over the respective subtrees.

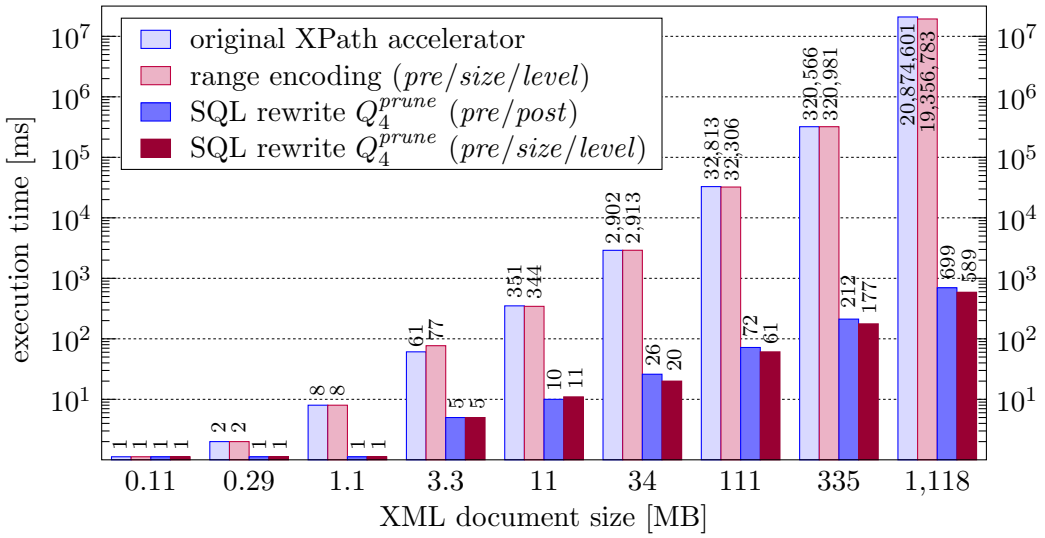


Figure 3.20: Context set pruning by a rewrite of the SQL code speeds up Query Q_4 by orders of magnitude (Q_4: `/descendant::city/following::zipcode`).

3.5 Related Work

An efficient access to the logical tree structure is crucial to the performance of any XML database system and since XML emerged in the database domain, a large body of research has been published on the efficient evaluation of path expressions. Quite surprisingly, though, most of these suggestions are limited to the child/descendant relationships.

3.5.1 Path Evaluation on RDBMSs

Staircase join is quite similar to the *multi-predicate merge join* (MPMGJN) developed by Zhang *et al.* [Zhang01]. In contrast to the traditional merge join (equi-joins only), this new algorithm handles *containment* predicates in a merge-like fashion. Containment in the sense of [Zhang01] precisely corresponds to the `descendant` and `child` axes in XPath. MPMGJN assumes a *pre*/*post*-like tree encoding and, hence, is directly applicable to XPath accelerator.

Much like staircase join, multi-predicate merge join is crafted for a high cache utilization and performs particularly well in that respect. The algorithm has been designed to exploit the hierarchical containment of intervals and, in contrast to staircase join, does not inspect further tree-specifics in the underlying tables.

A natural extension of the MPMGJN algorithm are the *structural joins* suggested by Al-Khalifa *et al.* [Al-Khalifa02]. They specifically target the evaluation

of location steps on XML tree structures and introduce additional support for the `ancestor` and `parent` axes.

The structural join algorithms inherit the skipping of irrelevant document regions from MPMGJN and guarantee a proper ordering of their output according to XML document order. They do not, however, employ staircase join's pruning technique. This makes them easier applicable to a loop-lifted step evaluation on the one hand, but may constitute a performance penalty on the other.

The *PathStack* algorithm proposed by Bruno *et al.* takes quite a different approach to path evaluation [Bruno02]. PathStack accepts an entire path pattern as its input and matches it against an XML document instance in a holistic fashion. Support for twig patterns is provided in terms of the accompanying *TwigStack* algorithm that divides the search for such twigs into several path queries (evaluated with PathStack). Results are then "stitched" together to form the overall result.

PathStack may significantly benefit from the existence of indexes on the stored document instances and, hence, interoperates well with existing database technology. In this proposal, indexes provide a *stream* of nodes that qualify for a specific node predicate in the query pattern. In a sense, this is comparable to the pushdown of name tests through staircase join and the respective application of indexes.

Though PathStack efficiently evaluates arbitrary path patterns with at most $|\mathsf{doc}|$ accesses to the document relation `doc`, its authors have already identified one of its drawbacks themselves. In contrast to the algorithms mentioned before, PathStack can only keep to XPath's ordering constraints if it is evaluated in a "blocking" fashion.

3.5.2 Tree Properties in XPath

Staircase join thoroughly exploits the semantics of XPath location steps for their efficient evaluation on *pre*/*post*-encoded data. Others have looked into similar characteristics of the query language, without the specific execution of queries on a relational back-end in mind.

The complexity of path step evaluation has been investigated by Gottlob *et al.* [Gottlob05]. Based on a reasonable evaluation strategy, they conclude an exponential time requirement in the query size for path expressions. This complexity is mainly a consequence of the growing size of *intermediate* query results. To exemplify, each repetition in the query

$$\texttt{//a/b}\underbrace{\texttt{/parent::a/b}}_{n \text{ times}}\ldots\texttt{/parent::a/b}$$

multiplies the intermediate result size by the number of `b` children below an `a` node. Staircase join's *pruning* phase minimizes the number of context nodes before

executing each step. Hence, our algorithm would *not* suffer from the effect sketched here. In fact, experiments confirmed a linear scaling for this example.

Our algorithm *does* suffer, however, during the evaluation of other test queries listed by Gottlob *et al.*. This is mainly due to the lack of *early-out semantics* in our implementation of staircase join. Experiments 2–4 in [Gottlob05] employ XPath predicate expressions that lead to exponential sizes of intermediate results even with staircase join.

The opportunity to *prune* nodes from an XPath location step context has been observed by Helmer *et al.* in connection with the algebraic optimizer included in Natix [Helmer02]. In Natix, pruning for the `preceding`, `following`, and `descendant` axes is carried out with the help of explicit *step functions*, with the same effect as sketched in Section 3.2.1.

Grust *et al.* have investigated the consequences of context pruning on *positional predicates* in XQuery [Grust04a]. It is easy to see that the removal of context nodes falsifies the query outcome in that case. As a remedy, the authors propose a refined implementation of staircase join that directly incorporates predicates on positions.

The preservation of *document order* in the result set of different location paths has been discussed by Fernández *et al.* [Fernández04]. Depending on sorting and uniqueness properties of the input to a location step, some XPath axes give similar guarantees for their evaluation result. Simplified reasonings of that kind have inspired our implementation of the non-recursive XPath axes (cf. Section 3.4.2).

4

Loop-Lifting: From XPath to XQuery

As we have seen, suitable encoding techniques can turn relational databases into highly efficient tree processors, particularly if their kernel has been explicitly enhanced for tree awareness, *e.g.*, in terms of the staircase join operator. These techniques, however, have remained focused on the execution of *XPath* queries so far. In this chapter, we will leverage our relational processing stack to *full XQuery support*, including the evaluation of `FLWOR` clauses with arbitrary nesting. This iteration primitive defined in the XQuery language specification will be handled in terms of the *loop-lifting* technique that we already touched upon in Chapter 3. It will form the key contribution of this chapter. We propose an XQuery compilation procedure that adheres to a relational algebra dialect easily implementable on, *e.g.*, SQL hosts.

Our assumptions about the relational back-end in our processing stack remain minimal. After setting up a consistent representation for XQuery's fundamental data type, *sequences of items*, we will identify our back-end requirements in Section 4.1. The loop-lifted compilation of XQuery `FLWOR` clauses will form the topic of Section 4.2, before we discuss additional XQuery features in Section 4.3. We will see that loop-lifted compilation goes perfectly together not only with the handling of XML tree nodes (Section 4.4), but also with the dynamic typing facilities in XQuery (Section 4.5). Experiments on a commercial SQL system in Section 4.6 confirm the applicability of our approach, before we wrap up in Section 4.7.

4.1 A Relational Algebra for XQuery

The set-oriented processing model of relational systems perfectly fits the set semantics of XPath location steps, which is of great advantage for the tree navigation performance we saw earlier. XQuery's tight ordering constraints on (sub-)expression results as well the means to iterate over them (`FLWOR` expressions) seem to be quite contrary to that and require specific care in the setup of our compiler. Hence, we will first have to draw our attention to an adequate representation of XQuery item sequences by relational means.

4.1.1 Relational Sequence Encoding

The XQuery language specification considers data of two principal kinds: XML *tree nodes* and *atomic values*. They are collectively referred to as *items* in the W3C XQuery Data Model (XDM) specification [Fernández05]. Items may heterogeneously be assembled into sequences and the evaluation of any XQuery (sub-) expression must result in an instance of XQuery's fundamental data type, *ordered, finite sequences of items*.

An important characteristic is the prohibition of nesting. XQuery sequences will be immediately *flattened* upon combination or concatenation. This blends perfectly with a relational representation as shown by the table on the left, which illustrates our encoding of the sequence `("a","b","c","d")`. In this relation, each tuple $\langle p, v\rangle$ encodes a single sequence item i, where p maintains the order among items (*i.e.*, the position in the sequence) and v carries the actual payload, *i.e.*, the value of i. We assume column *item* to be of a polymorphic type that may hold the *values* of items of atomic type as well as *node surrogates* for XML tree nodes (*e.g.*, their preorder ranks $pre(v)$). The polymorphic column types available in modern SQL implementations (*e.g.*, the `SQL_VARIANT` type in Microsoft SQL Server [SQL06]) are perfectly suited for this purpose.

pos	*item*
1	`"a"`
2	`"b"`
3	`"c"`
4	`"d"`

We encode the empty sequence `()` in terms of the empty relation. A single item i and the singleton sequence `(`i`)` have identical representations, which coincides with the XQuery semantics. We will usually populate column *pos* with intuitive, consecutive numbers $1, 2, 3, \ldots$. Our compilation procedure, however, does not actually depend on such a *dense* numbering in the relational encoding of a sequence.

Interfacing with XML Trees and Fragments

Our approach does not prescribe any specific tree encoding for the storage of XML instances. It only requires the availability of *node surrogates* γ that uniquely

identify each tree node. For a semantically correct XQuery translation, we assume these surrogates to correctly implement the basic XML concepts of *node identity* and *document order*, *i.e.*, for two nodes v_1, v_2 and their surrogates γ_{v_1}, γ_{v_2}, we require

$$v_1 \texttt{ is } v_2 \;\Leftrightarrow\; \gamma_{v_1} = \gamma_{v_2}$$

and

$$v_1 \texttt{ << } v_2 \;\Leftrightarrow\; \gamma_{v_1} < \gamma_{v_2} \;\;.$$

This requirement is satisfied by both encodings discussed in Chapter 2, in which node surrogates took the form of the nodes' preorder rank $pre(v)$. Operators `is` and `<<` then compile into integer comparisons on ranks. Other encodings serve our purpose equally well and may easily be plugged into the compilation procedure, *e.g.*, ORDPATH labels [O'Neil04] or extended preorder ranks [Li01].

XQuery is not limited to queries on single XML documents. In general, query evaluation involves nodes from multiple documents or fragments thereof, possibly created at runtime using XQuery's element constructors. To exemplify, the query

```
(element a { element b { () }}, element c { () })
```

creates three element nodes in two independent fragments. To capture this information by relational means, we extend the relational tree encoding by a new property *frag* and record a unique fragment identifier for each constructed fragment.

Figure 4.1 illustrates this concept for two XML fragments and the *range encoding* discussed in Section 2.1.7. For reasons that we will elaborate in Section 4.4.2, this encoding is a particularly good fit for the compilation approach pursued here and, hence, will be the encoding of choice for the remainder of this chapter. Note that we kept the document order of two nodes v_1 and v_2 from separate fragments consistent with the XQuery semantics: if v_1 precedes v_2 (*i.e.*, $v_1 \texttt{ << } v_2 \equiv pre(v_1) < pre(v_2)$), the same is true for any pair of nodes taken from these two fragments.

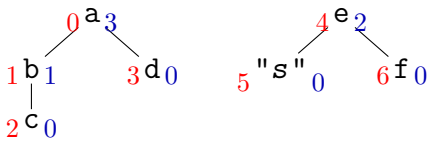
Loop-Lifting for Constant Subexpressions

Our prime concern in this chapter is the sound implementation of XQuery's *iteration* primitive, the `for-return` construct. In a nutshell, a `for` expression successively binds a variable `$`v to the items listed in the clause's `in` part. The `return`

```
<a>
  <b><c/></b>
  <d/>
</a>
```

```
<e>
  s<f/>
</e>
```

(a) Two XML fragments.



(b) Associated fragment trees.

pre	*size*	*level*	*kind*	*prop*	*frag*
0	3	0	elem	a	0
1	1	1	elem	b	0
2	0	2	elem	c	0
3	0	1	elem	d	0
4	2	0	elem	e	1
5	0	1	text	s	1
6	0	1	elem	f	1

(c) Relational encoding.

Figure 4.1: The addition of column *frag* to the relational tree encoding implements the distinctness of independent XML tree fragments. This chapter assumes documents stored according to the range encoding (*pre*/*size*/*level*).

body e is then evaluated for each item and its sub-results are assembled to form the overall expression result:

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

($e[x/\$v]$ denotes the consistent replacement of all free occurrences of $\$v$ in e by x).

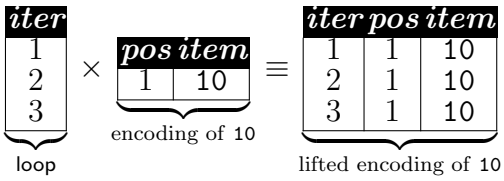
The semantics of these `FLWOR` clauses remains purely functional: it is sound to evaluate e for all n bindings of $\$v$ in parallel. This property suggests that we encode *all* bindings of $\$v$ within the loop body e in a *single* relation, which forms the basis for our loop-lifted compilation strategy:

(i) We represent a loop of n iterations by means of a relation loop with a single column *iter* of n values (*e.g.*, $1, 2, \ldots, n$).

(ii) If a subexpression e occurs inside the body of an XQuery `FLWOR` clause, its relational representation is *lifted* with respect to loop (intuitively, this accounts for the n independent evaluations of the loop body).

For a constant atomic value c, *lifting* with respect to a given loop relation means to form the Cartesian product

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



(a) Lifting the constant `10`.

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

(b) Lifted encoding of `(10, 20)`.

Figure 4.2: Loop-lifting for constant subexpressions. Lifted encodings for expressions $c =$ `10` and $c =$ `(10, 20)` with respect to the loop `for` $\$v_0$ `in (1, 2, 3) return` c.

Figure 4.2(a) shows an example of loop-lifting for the constant subexpression `10`, lifted with respect to the loop

`for` $\$v_0$ `in (1, 2, 3) return 10` .

If, for example, we replace `10` by the sequence `(10, 20)`, the loop-lifted representation consists of the six tuples shown in Figure 4.2(b) instead.

To ensure compositionality for arbitrary nestings of XQuery `FLWOR` clauses, we will use the $\langle iter, pos, item\rangle$ schema throughout the whole compilation procedure. In this representation of a subexpression e, a tuple $\langle i, p, v\rangle$ may be read as the assumption that, during the iteration identified by i, the item at position p in e has value v. In the following, we will refer to this schema as the *loop-lifted* representation of the XQuery expression e.

Before we derive a complete compilation scheme that exclusively operates on loop-lifted sequence representations, we will briefly formalize the target language of our compiler, the relational algebra we use.

4.1.2 An Algebra for XQuery

Our compilation procedure uses a relational algebra dialect that consists of the operators lined up in Table 4.1. Most of these operators are rather standard or even restricted variants of the operators found in a classical relational algebra. For the join operator $\bowtie$, *e.g.*, it is sufficient to evaluate equality predicates only. The selection operator σ_a solely selects tuples with column $a =$ `true`. Such a column is typically the output of an operator $\circledcirc_{a:\langle b_1,\ldots,b_n\rangle}$ which applies the n-ary operator $\circ$ to columns $b_1, \ldots, b_n$ and extends the input tuples by the result column a.

We do not expect the underlying implementation to implicitly remove duplicates from computed relations. In particular, the union operator $\uplus$ will only be

$\pi_{a_1:b_1,\ldots,a_n:b_n}$	projection and column renaming
σ_a	select tuples with $a = \mathsf{true}$
$\dot\cup$, $\setminus$	disjoint union, difference
δ	duplicate elimination
$\times$	Cartesian product
$\bowtie_{a=b}$	equi-join
$\varrho_{a:\langle b_1,\ldots,b_n\rangle \| p}$	sorted row numbering (with ordering $\langle b_1, \ldots, b_n\rangle$, grouping p)
$\#_a$	unsorted (arbitrary) row numbering
$\varsigma_{\alpha,\nu}$	XPath step join (axis α, node test ν)
ε/τ	element/text node construction
$\circledcirc_{a:\langle b_1,\ldots,b_n\rangle}$	n-ary arithmetic/comparison/Boolean operator $\circ$
$\mathsf{grp}_{a:\circ\, b \| p}$	aggregation/grouping ($\circ \in \{\mathsf{min}, \mathsf{max}, \mathsf{sum}, \mathsf{count}, \ldots\}$)
	literal table

Table 4.1: A relational algebra of rather standard operators constitutes the target language of our compiler (a, b column names).

applied to *disjoint* arguments. Our compiler triggers any required duplicate removal with the explicit δ operator.

All in all, this forms a rather simplistic relational algebra and we will later benefit from its *assembly style* notation when it comes to effective algebraic optimizations and rewrites.

A Tribute to XQuery's Order Semantics: ϱ and

Though our compiler's target language operates on relations, an inherently unordered data model, we still demand strict adherence to XQuery language specifications, including its prevalent concept of *order*. To this end, the generated plans make frequent use of the two *row numbering* operators ϱ and #. Given the sort order defined by columns $b_1, \ldots, b_n$, operator $\varrho_{a:\langle b_1,\ldots,b_n\rangle \| p}$ produces consecutive row numbers in the new column a. Row numbers re-start at 1 for each partition defined by the optional grouping attribute p.

This type of row number assignment can induce a significant cost: a physical implementation of ϱ will typically involve a blocking—hence, expensive—sort of its input relation. If the actual order among assigned numbers does not matter to the semantics of the plan, we may thus better trade the *sorted* row numbering operator ϱ for its *unsorted* counterpart #, which simply enumerates tuples as it goes in arbitrary order. Both variants are readily provided by many RDBMS implementations. According to the OLAP amendment to the SQL:1999 standard,

e.g., compliant systems provide $\varrho_{a:\langle b_1,\ldots,b_n\rangle\|p}(q)$ in terms of [Melton03]

```
SELECT *, ROW_NUMBER() OVER (ORDER BY b1,...,bn PARTITION BY p) AS a
  FROM q .
```

Database hosts operating on ordered relations may even provide such numbering for free. We have already discussed the `void` columns in the MonetDB [Boncz02] RDBMS in Chapter 2. To cope with the unsorted numbering operator #, a system's internal *row identifier* could serve as an implementation of # free of charge. The exploitation of both opportunities is on our agenda in Chapter 5 where we discuss order-aware optimizations in Pathfinder.

Access to XML Trees and Fragments

To keep our compilation procedure independent of any specific tree encoding, we express access to XML tree nodes in terms of explicit operators that we assume to be provided by the underlying back-end. Operators ◿, ε, and τ, *e.g.*, describe the evaluation of XPath location steps and the construction of transient element/text nodes, respectively. On range-encoded data, the staircase join from Chapter 3 provides an efficient implementation for ◿, other means to evaluate path steps include TwigStack [Bruno02] or structural joins [Al-Khalifa02].[1] We will sketch the semantics as well as a possible implementation for ε and τ in Section 4.4.2.

The tree access operators are expected to operate on node surrogates γ as demanded earlier. Typically, these surrogates point into a database table `doc` of persistently stored XML documents or refer to XML tree fragments constructed at runtime. In order to have the respective node containers at hand whenever an algebraic operator requires access to it,[2] our compiler maintains a set of *live nodes* Δ associated with every XQuery (sub-)expression. Note that we will postpone a detailed discussion on live nodes to Section 4.4.1. For the time being, the reader may think of the maintenance of Δ as a simple form of data flow analysis for XML tree nodes.

4.1.3 A Ruleset to Compile XQuery

The core of our XQuery-to-relational algebra compiler is described in terms of *inference rules* that define the $\cdot \Mapsto \cdot$ (read "compiles to") function. In these rules, a judgment of the form

$$\Gamma; \mathsf{loop} \vdash e \Mapsto (q, \Delta)$$

reads that, given

[1] You may want to read the ◿ symbol as "step operator" there.

[2] This does not only apply to operators ◿, ε, and τ, but also, *e.g.*, to a post-processing procedure that serializes the overall query result back into XML.

(i) an environment Γ that provides a mapping for any variable $\$v$ that may occur free in e to its algebraic equivalent (q_v, Δ_v) and

(ii) a relation loop that describes the iteration context of e,

the XQuery (sub-)expression e compiles to the algebraic expression q with an associated set of live nodes Δ.

Our compiler is fully compositional: to translate any subexpression independently of its surrounding, both, in- and output, of the $\cdot \Mapsto \cdot$ function are represented in a loop-lifted manner, *i.e.*, they adhere to the schema $\langle iter, pos, item \rangle$. We invoke the compilation process with the top-level XQuery expression, an empty environment $\Gamma = \emptyset$, and a singleton loop relation ($\mathsf{loop} \equiv \boxed{\begin{array}{|c|}\hline \textbf{\textit{iter}} \\ \hline 1 \\ \hline \end{array}}$), which indicates that the top-level expression is not embedded in any iteration.

Our compilation process accepts the normalized *XQuery Core* dialect as its input, which basically represents a subset of the XQuery surface language, with most syntactic sugar removed. In accordance with the W3C Formal Semantics specification [Draper05], the Pathfinder XQuery compiler transforms incoming queries into XQuery Core before initiating the actual compilation. This transformation, *e.g.*, trades `where` clauses in XQuery `FLWOR` expressions for the respective conditional `if-then-else` and positional predicates for an explicit iteration over the context set.

The output of the compiler is a single algebra query and a set of live nodes (in the form of an algebraic expression, too) associated with it. The generated tuples will typically be consumed by a post processing step that serializes the query result back into XML.

4.1.4 Basic XQuery Expressions

Equipped with this means of notation, we can now formalize the loop-lifting idea that we have already seen for constant subexpressions.

Constant Subexpressions

The informal derivation of loop-lifted sequence representations (page 76) directly translates into the inference rule that handles literal values in XQuery:

$$\frac{}{\Gamma; \mathsf{loop} \vdash c \Mapsto \left(\mathsf{loop} \times \boxed{\begin{array}{|c|c|}\hline \textbf{\textit{pos}} & \textbf{\textit{item}} \\ \hline 1 & c \\ \hline \end{array}}, \emptyset\right)} \quad \text{(Const)}$$

Since no nodes can be contained in the expression result of a literal atomic value, the compiler output is associated with the empty set of live nodes $\emptyset$.

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

(a) Encoding q_1 of e_1.

iter	pos	item
1	1	2
2	2	30

(b) Encoding q_2 of e_2.

p	ord	iter	pos	item
1	1	1	1	1
2	2	1	1	2
1	1	2	1	10
2	1	2	2	20
3	2	2	1	30

(c) Temporary column attachment.

iter	pos	item
1	1	1
1	2	2
2	1	10
2	2	20
2	3	30

(d) Encoding of (e_1, e_2).

Figure 4.3: Sequence construction: a temporary *ord* column facilitates the establishment of the correct sequence order (temporary column p) over the expression result (cf. Rule SEQ).

4.1.5 Sequence Construction

Essentially, we compute the sequence construction operator (e_1, e_2) in terms of a *disjoint union* of the relational encodings q_1 and q_2 of e_1 and e_2:

$$\frac{\Gamma; \mathsf{loop} \vdash e_1 \Rightarrow (q_1, \Delta_1) \qquad \Gamma; \mathsf{loop} \vdash e_2 \Rightarrow (q_2, \Delta_2)}{\Gamma; \mathsf{loop} \vdash (e_1, e_2) \Rightarrow \left(\pi_{iter,pos:p,item}\left(\varrho_{p:\langle ord,pos\rangle \| iter}\left(\left(\boxed{\begin{array}{c}\textit{ord}\\1\end{array}} \times q_1\right) \cup \left(\boxed{\begin{array}{c}\textit{ord}\\2\end{array}} \times q_2\right)\right)\right), \Delta_1 \cup \Delta_2\right)} \quad (\textsc{Seq})$$

To ensure the correct logical ordering of the expression result (items coming from e_1 must appear before those from e_2 in *pos* order), we temporarily extend both operands q_1 and q_2 by a column *ord*, encoding the order of the arguments. After the union, Rule SEQ sets up the final *pos* column (named p first to avoid name clashes) using the renumbering operator $\varrho_{p:\langle ord,pos\rangle \| iter}$, before the projection operator π restores the $\langle iter, pos, item\rangle$ schema.

Note that this approach parallelly evaluates the sequence construction for *all* iterations encoded in q_1 and q_2 at once. Figure 4.3 demonstrates how the resulting algebra code is evaluated. Relation q_1 (Figure 4.3(a)) encodes two sequences in two iterations: the singleton sequence `(1)` in iteration 1 and the two-item sequence `(10, 20)` in the second iteration. The second argument of the sequence constructor are the two singletons `(2)` and `(30)` in iterations 1 and 2, respectively, as displayed in Figure 4.3(b). The disjoint union followed by the renumbering operator in Rule SEQ yields the intermediate table shown in Figure 4.3(c). The overall expression result after column projection/renaming is the relation in Figure 4.3(d).

Note that the "append union" that systems typically employ to evaluate $\cup$ will lead to a costly sort of the intermediate result to implement row numbering. Assuming properly sorted input relations, however, we would be much better off evaluating $\cup$ as a "merge union", which rendered the subsequent ϱ into a no-cost

operation. We will have a closer look at optimizations of this kind in the next chapter.

Variable Binding and Usage

We handle variable bindings expressed through XQuery's `let` construct entirely at query compilation time: to compile `let` $\$v$ `:=` e_1 `return` e_2, we translate e_1 in the currently active context (consisting of the environment Γ and relation loop) to yield the algebra code q_1 and then use the enriched environment $\Gamma + \{\$v \mapsto (q_1, \Delta_1)\}$ to compile e_2:

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

A reference to a variable $\$v$ then simply compiles to a lookup in Γ:

$$\frac{}{\{\ldots, \$v \mapsto (q_v, \Delta_v), \ldots\};\mathsf{loop} \vdash \$v \Rightarrow (q_v, \Delta_v)}\ . \qquad \text{(Var)}$$

Note that the maintenance of Γ is entirely performed at *compile time*. Effectively, this *unfolds* any `let` binding during the compilation process via Rules Let and Var. Unused bindings will vanish from the compilation result without any additional treatment. Such a strategy is likely to violate the XQuery semantics if node constructors appear in the binding expression e_1 of a `let` clause. In Section 4.4.3, we will see why our compiler guarantees standards-compliance nevertheless.

We will now turn to the heart of the loop-lifting compilation strategy, the translation of XQuery's `FLWOR` clauses.

4.2 Relational `FLWOR`s

The iteration primitive `for-return` constitutes a vital part of the XQuery language. The arbitrary nesting of these `FLWOR` clauses introduces a notion of *scopes* into the language. For instance, the query

$$\begin{array}{l} s\Big\{\ \texttt{(for } \$v_0 \texttt{ in } e_0 \texttt{ return} \\ \quad s_0\big\{\ \ e_0'\ \texttt{,} \\ \qquad \texttt{for } \$v_1 \texttt{ in } e_1 \texttt{ return} \\ \quad s_1\Big\{\ \ \texttt{for } \$v_1 \texttt{ in } e_{1\cdot 0} \texttt{ return} \\ \qquad s_{1\cdot 0}\big\{\ \ e_{1\cdot 0}' \\ \quad \texttt{)}\ \texttt{,} \end{array} \qquad (Q_1)$$

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

(a) q((1,2,3)).

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

(b) q_0($\$v_0$).

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

(c) q_0(10).

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

(d) q_0((10,$\$v_0$)).

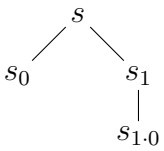
iter	pos	item
1	1	10
1	2	1
1	3	10
1	4	2
1	5	10
1	6	3

(e) Result in s.

Figure 4.4: Relational representation of intermediate results in Query Q_2.

defines four *variable scopes* as indicated by the curly braces. Variable $\$v_0$ is visible in scope s_0, variable $\$v_1$ in scopes s_1 and $s_{1.0}$, and $\$v_{1.0}$ may be accessed in scope $s_{1.0}$ only. No variables are bound in the top-level scope s in this example. These scopes coincide with the *iteration scopes* introduced by the respective `for` loops. The iteration over expressions found in scope s_0, *e.g.*, is determined by the bindings of variable v_0.

It is important to see that `FLWOR` clauses may be nested in an arbitrary fashion. In general, this leads to a tree-shaped hierarchy of scopes, as illustrated on the right for Query Q_1. We will use $s_{x \cdot y}$, $x \in \{0, 1, \dots\}^*$, $y \in \{0, 1, \dots\}$ in the following to identify the yth child of scope s_x. The loop-lifted representation of any expression result e is determined by the iteration scope s_x it appears in and we will use $q_x(e)$ to denote it. It is the task of the upcoming compilation rules (the back-mapping step in Section 4.2.4 in particular) to keep all relational representations consistent with their respective iteration scopes.



4.2.1 `for`-Bound Variables

To illustrate the consequences of our loop-lifting idea on XQuery `FLWOR` clauses, consider the query

$$s \left\{ \begin{array}{l} \texttt{for \$}v_0 \texttt{ in (1, 2, 3) return} \\ s_0 \{ \quad \texttt{(10, \$}v_0\texttt{)} \quad . \end{array} \right. \qquad (Q_2)$$

Figure 4.4 lines up the relational representations encountered for different subexpressions during the evaluation of this query. The top-level scope s is iterated only once, leading to a constant *iter* column in the relational representation of the sequence `(1, 2, 3)` in this scope (q((1, 2, 3)) in Figure 4.4(a)).

We now want to successively bind variable $\$v_0$ to the items in this sequence, *i.e.*, $\$v_0$ describes a singleton sequence in three different iterations. The relation in

Figure 4.4(b) accounts for this fact with a constant *pos* column and three distinct values in column *iter*. We can easily compute the representation of $\$v_0$ as follows:

$$q_0(\$v_0) \equiv \begin{array}{c} \boxed{\begin{array}{|c|}\hline \textbf{\textit{pos}} \\ \hline 1 \\ \hline \end{array}} \quad \times \\ \pi_{iter:inner,item} \\ \varrho_{inner:\langle iter,pos\rangle} \\ q(\texttt{(1,2,3)}) \end{array} . \tag{4.1}$$

We retain the values of the binding sequence `(1, 2, 3)` (to which $\$v_0$ shall be bound successively). To establish a new *iter* column with a consecutive numbering, we use the row numbering operator ϱ. Column *pos* is then filled with a constant value of 1, which we express as a Cartesian product with the singleton relation $\boxed{\begin{array}{|c|}\hline \textbf{\textit{pos}} \\ \hline 1 \\ \hline \end{array}}$.

The construction of the sequence `(10, `$\$v_0$`)` in scope s_0 then yields the representation shown in Figure 4.4(d). A back-mapping step (see Section 4.2.4) leverages the result into the top-level scope s (Figure 4.4(e)), establishing the overall query result.

The General Case

To generalize this idea, consider a `for`-loop in its directly enclosing scope s_x:

$$s_x \left\{ \begin{array}{l} \vdots \\ \texttt{for } \$v_{x\cdot y} \texttt{ in } e_{x\cdot y} \texttt{ return} \\ s_{x\cdot y} \left\{ \; e'_{x\cdot y} \right. \\ \vdots \end{array} \right. \tag{Q_3}$$

Again, we derive $q_{x\cdot y}(\$v_{x\cdot y})$, the representation of the iteration variable in the inner scope, from the representation $q_x(e_{x\cdot y})$ of its binding sequence $e_{x\cdot y}$ and employ operators ϱ and $\times$ for that task:

$$q_{x\cdot y}(\$v_{x\cdot y}) \equiv \begin{array}{c} \boxed{\begin{array}{|c|}\hline \textbf{\textit{pos}} \\ \hline 1 \\ \hline \end{array}} \quad \times \\ \pi_{iter:inner,item} \\ \varrho_{inner:\langle iter,pos\rangle} \\ q_x(e_{x\cdot y}) \end{array} . \tag{4.2}$$

Positional Variables

Optionally, a second variable $\$p_{x\cdot y}$ may be bound to the *position* of the iteration variable $\$v_{x\cdot y}$ within its binding sequence $e_{x\cdot y}$:

$$s_x \left\{ \begin{array}{l} \vdots \\ \texttt{for } \$v_{x\cdot y} \texttt{ at } \$p_{x\cdot y} \texttt{ in } e_{x\cdot y} \texttt{ return} \\ s_{x\cdot y} \left\{ \; e'_{x\cdot y} \right. \\ \vdots \end{array} \right.$$

With the help of the ϱ operator, we can easily establish the relational representation of such a positional variable based on the binding sequence $e_{x\cdot y}$: replace column *item* with a consecutive numbering within each *iter* group, then set columns *iter* and *pos* as described above for the iteration variable $\$v_{x\cdot y}$:

$$q_{x\cdot y}(\$p_{x\cdot y}) \equiv \begin{array}{c} \times \\ \boxed{\begin{array}{c}\textbf{\textit{pos}}\\ 1\end{array}} \quad \pi_{iter:inner,item} \\ \varrho_{inner:\langle iter,pos\rangle} \\ \varrho_{item:\langle pos\rangle \| iter} \\ \pi_{iter,pos} \\ q_x(e_{x\cdot y}) \end{array} . \tag{4.3}$$

We make both, the iteration variable $\$v_{x\cdot y}$ and the positional variable $\$p_{x\cdot y}$, accessible to the compilation of the loop body $e_{x\cdot y'}$ (see Rule VAR) by adding them to the variable environment Γ before compiling $e'_{x\cdot y}$.

4.2.2 Maintaining loop

The concept of loop-lifting requires the consistent maintenance of a relation loop, containing the independent iterations of the current scope. The semantics of the `for` construct prescribes the evaluation of its inner scope $s_{x\cdot y}$ once for each binding of the iteration variable $\$v_{x\cdot y}$. Based on the encoding of the latter, $q_{x\cdot y}(\$v_{x\cdot y})$, we define a new loop relation to compile the expression's `return` body $e'_{x\cdot y}$:

$$\mathsf{loop}_{x\cdot y} \equiv \begin{array}{c} \pi_{iter} \\ q_{x\cdot y}(\$v_{x\cdot y}) \end{array} . \tag{4.4}$$

This kind of loop relation was used to derive the loop-lifted representation of the constant `10` in earlier examples (cf. Figures 4.2(a) and 4.4(c)).

4.2.3 Free Variables in the `return` Clause

A variable introduced by means of an XQuery `FLWOR` clause in scope s_x will be accessible in any scope $s_{x\cdot x'}$, $x' \in \{0, 1, \dots\}^+$ enclosed by s_x. If scope $s_{x\cdot x'}$ is viewed in isolation, such variables appear to be free.

As the compiled representation of any expression depends on the iteration scope it appears in, any variable $\$w$ accessible in scope s_x must be made *compatible* with the enclosed scope $s_{x\cdot y}$ upon compilation of the `FLWOR` clause Q_3. We will now derive $q_{x\cdot y}(\$w)$ ($\$w$'s representation in scope $s_{x\cdot y}$) from its relational encoding in the enclosing scope s_x. To understand this derivation, consider the following

iter	pos	item
1	1	1
2	1	2

(a) $q_0(\$a)$.

iter	pos	item
1	1	1
2	1	1
3	1	2
4	1	2

(b) $q_{0\cdot0}(\$a)$.

iter	pos	item
1	1	10
2	1	20
3	1	10
4	1	20

(c) $q_{0\cdot0}(\$b)$.

outer	inner
1	1
1	2
2	3
2	4

(d) $\mathsf{map}_{(0,0\cdot0)}$.

Figure 4.5: Scope-dependent representation of variables $\$a$ and $\$b$ (Query Q_4). The connection between iteration scopes s_0 and $s_{0\cdot0}$ is captured by relation $\mathsf{map}_{(0,0\cdot0)}$.

evaluation of two nested `for` loops (note the reference to $\$a$ in the inner scope $s_{0\cdot0}$):

$$
\begin{array}{l}
s\ \{\\
\quad s_0\ \{\\
\quad\quad s_{0\cdot0}\ \{\\
\end{array}
\begin{array}{l}
\texttt{for \$}a\texttt{ in (1, 2) return}\\
\quad\texttt{(\$}a\texttt{,}\\
\quad\quad\texttt{for \$}b\texttt{ in (10, 20) return}\\
\quad\quad\quad\texttt{(\$}a\texttt{, \$}b\texttt{)}\\
\quad\texttt{)}
\end{array}
\qquad (Q_4)
$$

The first iteration of the outer loop binds $\$a$ to `1`. Two evaluations of the innermost scope $s_{0\cdot0}$ correspond to this iteration, with $\$b$ being successively bound to `10` and `20`. The second iteration of the outer loop involves another two evaluations of the innermost loop body, both with $\$a$ bound to `2`. In effect, scope $s_{0\cdot0}$ is iterated four times. Figures 4.5(a)–(c) illustrate the required relations for variables $\$a$ and $\$b$. Note how Figures 4.5(b) and 4.5(c) reflect that in iteration 3 of the inner loop body variable $\$a$ is bound to `2` while $\$b$ is bound to `10`, as desired.

The semantics of this nested iteration is captured by the relation $\mathsf{map}_{(0,0\cdot0)}$ shown in Figure 4.5(d), where an entry $\langle o, i\rangle$ indicates that during iteration i of the inner loop body in scope $s_{0\cdot0}$ the outer body (scope s_0) is in iteration o. In Figure 4.5(d), *e.g.*, iterations 1 and 2 of the inner loop body (scope $s_{0\cdot0}$) correspond to the first iteration over the outer scope s_0; iterations 3 and 4 in $s_{0\cdot0}$ occur within the second iteration over s_0.

More generally, we introduce $\mathsf{map}_{(x,x\cdot y)}$ to relate iteration identifiers of a scope $s_{x\cdot y}$ to those of its directly enclosing scope s_x. The iteration identifiers of scope $s_{x\cdot y}$ originate from the derivation of $q_{x\cdot y}(\$v_{x\cdot y})$, the relational encoding of the iteration variable $\$v_{x\cdot y}$ (see Equation 4.2). Analogously to the derivation of $q_{x\cdot y}(\$v_{x\cdot y})$, we compute relation $\mathsf{map}_{(x,x\cdot y)}$ based on $q_x(e_{x\cdot y})$, the relational representation of the

binding sequence $e_{x\cdot y}$:

$$\mathsf{map}_{(x,x\cdot y)} \equiv \pi_{outer:iter,inner} \circ \varrho_{inner:\langle iter,pos\rangle} \circ q_x(e_{x\cdot y}) \quad . \tag{4.5}$$

With this connection between enclosing iteration scopes at hand, we can now *map* variables to the inner scope of a `for`-loop and derive the representation of any free variable $\$w$ in scope $s_{x\cdot y}$ by means of an equi-join:

$$q_{x\cdot y}(\$w) \equiv \pi_{iter:inner,pos,item} \circ \bowtie_{iter=outer} (q_x(\$w), \mathsf{map}_{(x,x\cdot y)}) \quad . \tag{4.6}$$

To make this representation available during the compilation of the `for`-body, we insert the binding $\$w \mapsto q_{x\cdot y}(\$w)$ into the variable environment Γ used to compile the body (along with the bindings for the iteration and positional variables, $\$v_{x\cdot y}$ and $\$p_{x\cdot y}$). Rule VAR will then establish compatible representations for each occurrence of a reference to variable $\$w$.

4.2.4 Mapping Back

Based on the relational representations of variables $\$a$ and $\$b$, the system is now ready to compute the sequence construction `($a, $b)` in the innermost loop body (scope $s_{0\cdot 0}$) of Query Q_4, yielding the table shown in Figure 4.6(a). Note, however, that the *second* sequence construction operation `($a, for $b in ...)` in this query requires this intermediate result to be compatible with the enclosing scope s_0 (*i.e.*, as illustrated in Figure 4.6(b)). Hence, we need to map $q_{0\cdot 0}$(`($a, $b)`) back into s_0.

In a sense, this *back-mapping* step implements the assembly of all item sequences from independent loop iterations into a single result sequence. Relation $\mathsf{map}_{(x,x\cdot y)}$ again provides the necessary connection between two directly enclosing scopes s_x and $s_{x\cdot y}$ and an equi-join with $\mathsf{map}_{(x,x\cdot y)}$ retrieves the required iteration identifiers from the parent scope. A subsequent ϱ operator brings all result items into correct order:

$$\pi_{iter:inner,pos:pos_1,item} \circ \varrho_{pos_1:\langle iter,pos\rangle \| outer} \circ \bowtie_{iter=inner} (q_{x\cdot y}(e), \mathsf{map}_{(x,x\cdot y)}) \quad .$$

This delivers the missing pieces to complete the evaluation of Query Q_4. Figure 4.6 lines up the intermediate results involved: the result of back-mapping the

iter	*pos*	*item*
1	1	1
1	2	10
2	1	1
2	2	20
3	1	2
3	2	10
4	1	2
4	2	20

(a) $q_{0\cdot0}$((`$a`,`$b`)).

iter	*pos*	*item*
1	1	1
1	2	10
1	3	1
1	4	20
2	1	2
2	2	10
2	3	2
2	4	20

(b) Inner loop in s_0.

iter	*pos*	*item*
1	1	1
2	1	2

(c) $q_0(\$a)$.

iter	*pos*	*item*
1	1	1
1	2	1
1	3	10
1	4	1
1	5	20
1	6	2
1	7	2
1	8	10
1	9	2
1	10	20

(d) Final result in s.

Figure 4.6: Intermediate and final results during the evaluation of Query Q_4.

inner loop body into scope s_0 (Figure 4.6(b)), made compatible with the representation of variable $\$a$ in scope s_0 (Figure 4.6(c)), and, finally, the overall query result after mapping back the sequence construction result into the top-level scope (Figure 4.6(d)).

4.2.5 Complete Compilation Rule for `FLWOR` Expressions

Inference rule For lines up the complete translation of XQuery `for` loops, also ensuring the proper maintenance and propagation of the loop relation through the compilation process:

$$
\begin{array}{ll}
① & \{\ldots, \$v_i \mapsto (q_{v_i}, \Delta_{v_i}), \ldots\}; \mathsf{loop} \vdash e_1 \Rightarrow (q_1, \Delta_1) \\
② & q_v \equiv \boxed{\begin{array}{c}\textit{pos}\\ 1\end{array}} \times \pi_{iter:inner,item}\left(\varrho_{inner:\langle iter,pos\rangle}(q_1)\right) \\
③ & q_p \equiv \boxed{\begin{array}{c}\textit{pos}\\ 1\end{array}} \times \pi_{iter:inner,item}\left(\varrho_{inner:\langle iter,pos\rangle}\left(\varrho_{item:\langle pos\rangle \| iter}\left(\pi_{iter,pos}(q_1)\right)\right)\right) \\
④ & \mathsf{loop}_v \equiv \pi_{iter}(q_v) \qquad \mathsf{map} \equiv \pi_{outer:iter,inner}\left(\varrho_{inner:\langle iter,pos\rangle}(q_1)\right) \\
⑤ & \Gamma_v \equiv \{\ldots, \$v_i \mapsto (\pi_{iter:inner,pos,item}\left(q_{v_i} \bowtie_{iter=outer} \mathsf{map}\right), \Delta_{v_i}), \ldots\} \\
 & \quad + \{\$v \mapsto (q_v, \Delta_1)\} + \{\$p \mapsto (q_p, \emptyset)\} \\
⑥ & \Gamma_v; \mathsf{loop}_v \vdash e_2 \Rightarrow (q_2, \Delta_2) \\
\hline
⑦ & \{\ldots, \$v_i \mapsto (q_{v_i}, \Delta_{v_i}), \ldots\}; \mathsf{loop} \vdash \texttt{for}\ \$v\ \texttt{at}\ \$p\ \texttt{in}\ e_1\ \texttt{return}\ e_2 \Rightarrow \\
 & \left(\pi_{iter:outer,pos:pos_1,item}\left(\varrho_{pos_1:\langle inner,pos\rangle \| outer}\left(q_2 \bowtie_{iter=inner} \mathsf{map}\right)\right), \Delta_2\right)
\end{array}
\quad \text{(For)}
$$

The compilation procedure captured by this rule may be summarized as follows:

① Compile the binding expression e_1 in the current context.

② Derive the relational representation of the iteration variable $\$v$.

③ If present in the query text, establish the representation of the positional variable $\$p$.

④ Set up relations loop and map to prepare the compilation of the loop body e_2.

⑤ The new variable environment Γ for the compilation of e_2 consists of all variables currently free (after mapping them to the scope of the loop body), the iteration variable $\$v$, and (optionally) the positional variable $\$p$.

⑥ In this new context, invoke the compilation of the loop body e_2.

⑦ Finally, map the compilation result of e_2 back into the parent scope using relation map.

4.2.6 Optional: The `order by` Clause

Rule For above will concatenate the result sequences for individual evaluations of the `return` body in the order prescribed by the binding sequence e_1 to form the overall `FLWOR` expression result. Optionally, a user may explicitly request a different arrangement of all subsequences by stating an `order by` clause. The results of the independent evaluations of e_2 in the loop

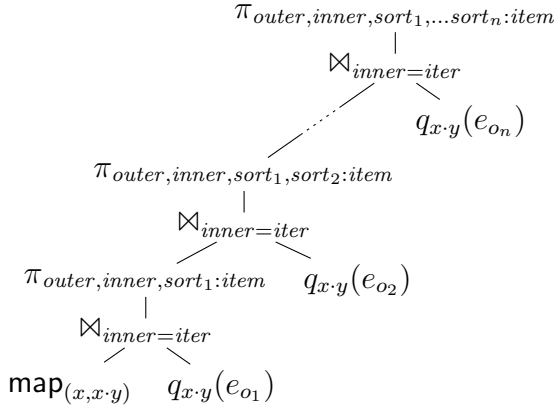
$$\texttt{for } \$v \texttt{ at } \$p \texttt{ in } e_1 \texttt{ order by } e_{o_1}\texttt{, } e_{o_2}\texttt{, } \ldots\texttt{, } e_{o_n} \texttt{ return } e_2 \ ,$$

for example, will be concatenated according to the lexicographic order described by expressions $e_{o_1}, \ldots, e_{o_n}$ (with major ordering on e_{o_1}; expressions e_{o_i} will be evaluated for each binding of $\$v$). Note that this only affects the *assembly* of the `return` clause's result sequences, but *not* the outcome of individual evaluations of e_2. In particular, the positional variable $\$p$ still indicates the association of the binding variable $\$v$ with its position within the binding sequence e_2.

It is the back-mapping step ⑦ that implements the assembly of result sequences in Rule For. With only a slight extension to the map relation, we may seamlessly integrate the handling of `order by` clauses into our compilation process. We translate all sort specifiers e_{o_i} analogously to the compilation of the loop body e_2 itself and extend the relation map to collect the *item* columns of all e_{o_i}, yielding

the relation $\mathsf{map}'_{(x,x\cdot y)}$ with schema $\langle outer, inner, sort_1, \ldots, sort_n\rangle$:[3]

$$\mathsf{map}'_{(x,x\cdot y)} \equiv \quad (4.7)$$



Based on map', we can now derive the result's position information (column *pos*) in consistence with the `order by` clause. The result of the back-mapping step ⑦ in Rule For then reads (note the modified parameters to ϱ):

$$\pi_{iter:outer,pos:pos_1,item}\left(\varrho_{pos_1:\langle sort_1,\ldots,sort_n,pos\rangle\|outer}\left(q_2 \bowtie_{iter=inner} \mathsf{map}'\right)\right) \quad .$$

4.3 Other Expression Types

To exemplify the loop-lifting concept for arbitrary XQuery Core constructs, we will focus on arithmetics and conditionals (`if-then-else` clauses) in this section. Section 4.4 will then discuss our support for operations on XML tree nodes (path navigation and element construction), before we round up the compilation procedure with a glimpse into an implementation of XQuery's dynamic type semantics (Section 4.5).

4.3.1 Arithmetics/Comparisons

During the compilation process, we map arithmetic operators (`+`, `*`, ...) as well as XQuery's value comparison operators (`eq`, `gt`, ...) to their readily available implementation on the underlying back-end ($\oplus$, $\circledast$, $\ominus$, $\oslash$, ...). The relational $\circledcirc$ operators evaluate their XQuery counterpart for all iterations at once. We use the

[3] This algebra code assumes expressions e_{o_i} to be of type `xs:anyAtomicType`. The less restrictive type `xs:anyAtomicType?` [Boag05, § 3.8.3] is easily covered if the equi-joins in (4.7) are replaced by left outer joins and the underlying database can handle null values consistent with XQuery's `empty greatest` or `empty least` specifiers.

following plan to evaluate the XQuery expression $e_1 \circ e_2$:

$$q(e_1 \circ e_2) \equiv \pi_{iter,pos,item:res}(\circledcirc_{res:(item,item')}(q_1 \bowtie_{iter=iter'} (\pi_{iter':iter,item':item}(q_2)))) \quad . \tag{4.8}$$

The relational join over *iter* columns pairs tuples that originate from the same iteration (column renaming with π ensures uniqueness of column names beforehand). Operator $\circledcirc$ then attaches the new column *res* with the result of the arithmetic/comparison operation. Finally, column projection and renaming establish the $\langle iter, pos, item \rangle$ schema used throughout the compilation.

Compilation rule ARITH below expresses this translation in our notation of inference rules (note that expressions e_i cannot evaluate to nodes, hence, are associated with empty live node sets):

$$\frac{\Gamma; \mathsf{loop} \vdash e_1 \Rightarrow (q_1, \emptyset) \quad \Gamma; \mathsf{loop} \vdash e_2 \Rightarrow (q_2, \emptyset) \quad \circ \in \{\texttt{+}, \texttt{*}, \texttt{eq}, \texttt{gt}, \dots\}}{\Gamma; \mathsf{loop} \vdash e_1 \circ e_2 \Rightarrow \left(\pi_{iter,pos,item:res}\left(\circledcirc_{res:(item,item')}\left(q_1 \bowtie_{iter=iter'} \left(\pi_{iter':iter,item':item}(q_2)\right)\right)\right), \emptyset\right)} \quad . \tag{ARITH}$$

Observe that the equi-join to combine tuples with matching *iter* values will drop any iteration where either operand evaluates to the empty sequence (*i.e.*, where no tuple with the respective *iter* value exists in the relational encoding). This elegantly complies with the XQuery semantics [Boag05] which demands the result to be the empty sequence if either operand is the empty sequence.

4.3.2 Conditionals: `if-then-else`

To illustrate the loop-lifted evaluation of `if-then-else` clauses, consider the query

```
for $v in (3, 4, 5, 6) return
  if ($v mod 2 eq 0) then "even" else "odd" ,        (Q5)
```

(e_1: `($v mod 2 eq 0)`, e_2: `"even"`, e_3: `"odd"`)

which is to return the sequence `("odd","even","odd","even")`. Figure 4.7 lines up an evaluation trace of the compiled relational plan:

① Use the current context to compile the condition e_1. The resulting relational encoding for our example is the table labeled e_1 in Figure 4.7 (along with its

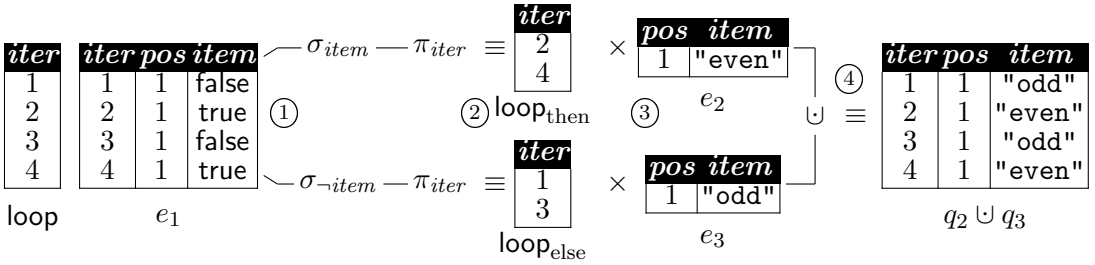


Figure 4.7: Evaluation trace for Query Q_5. Rule If establishes new loop relations for the evaluation of the `then` and `else` bodies. Both bodies employ a Cartesian product (Rule Const), subject to a disjoint union for the conditional's result.

corresponding loop relation).[4]

② Based on the outcome of the conditional expression e_1, establish new loop relations ($\mathsf{loop}_{\text{then}}$ and $\mathsf{loop}_{\text{else}}$ in Figure 4.7) and variable environments Γ (not applicable to our example) for the compilation of the `then` and `else` bodies e_2 and e_3.

③ Compile bodies e_2 and e_3 using the respective loop relations. In Query Q_5, both branches are constant subexpressions and thus compile into a cross product between loop and their relational representation (cf. Rule Const).

④ The disjoint union of the resulting relations represents the conditional's overall result, as illustrated on the right of Figure 4.7 for our example query.

Rule If captures this procedure in a single inference rule for `if-then-else` clauses (circled numbers refer to the processing steps sketched above):

$$\frac{\begin{array}{c}
① \quad \{\ldots, \$v_i \mapsto (q_{v_i}, \Delta_{v_i}), \ldots\}; \mathsf{loop} \vdash e_1 \Rightarrow (q_1, \emptyset) \\
② \quad \mathsf{loop}_{\text{then}} \equiv \pi_{iter}(\sigma_{item} q_1) \qquad \mathsf{loop}_{\text{else}} \equiv \pi_{iter}(\sigma_{\neg item} q_1) \\
② \; \Gamma_{\text{then}} \equiv \{\ldots, \$v \mapsto (\pi_{iter,pos,item}(q_{v_i} \bowtie_{iter=iter'} (\pi_{iter':iter}(\mathsf{loop}_{\text{then}}))), \Delta_{v_i}), \ldots\} \\
② \; \Gamma_{\text{else}} \equiv \{\ldots, \$v \mapsto (\pi_{iter,pos,item}(q_{v_i} \bowtie_{iter=iter'} (\pi_{iter':iter}(\mathsf{loop}_{\text{else}}))), \Delta_{v_i}), \ldots\} \\
③ \quad \Gamma_{\text{then}}; \mathsf{loop}_{\text{then}} \vdash e_2 \Rightarrow (q_2, \Delta_2) \qquad \Gamma_{\text{else}}; \mathsf{loop}_{\text{else}} \vdash e_3 \Rightarrow (q_3, \Delta_3)
\end{array}}{④ \quad \{\ldots, \$v_i \mapsto (q_{v_i}, \Delta_{v_i}), \ldots\}; \mathsf{loop} \vdash \texttt{if (}\, e_1 \,\texttt{) then}\; e_2 \;\texttt{else}\; e_3 \Rightarrow (q_2 \cup q_3, \Delta_2 \cup \Delta_3)} \quad \text{(If)}$$

The condition e_1 evaluates to a single item of type `xs:boolean` for all iterations. Hence, its algebraic equivalent q_1 must be associated with an empty set of live

[4] The normalized XQuery Core code will implicitly infer the condition's *effective boolean value* first in terms of `fn:boolean()` [Draper05, § 4.10]. In effect, the result of e_1 will contain exactly one item for each iteration in loop.

nodes $\emptyset$. The results of e_2 and e_3, in contrast, both contribute to the overall result. Thus, we associate the union of their live node sets, $\Delta_2 \cup \Delta_3$, with the overall result.

4.4 Interfacing with XML/XPath

So far, we have addressed most of the basic building blocks of the XQuery language specification, including the consistent handling of `for` iteration primitives. In this section, we will look into the integration of node-related features, *i.e.*, the evaluation of XPath location steps (Section 4.4.1) and XQuery's node construction facilities (Sections 4.4.2 and 4.4.3).

4.4.1 XPath Location Steps

The loop-lifting strategy is complementary to the evaluation of XPath. Efficient means to handle XPath have been covered in Chapters 2 and 3, though other algorithms could be plugged in equally well into our compilation procedure, including those described by Bruno *et al.* [Bruno02] and Al-Khalifa *et al.* [Al-Khalifa02]).

We encapsulate the evaluation of XPath location steps into the operator ◰ that we assume to be appropriately supported by the underlying back-end (*e.g.*, in terms of the aforementioned algorithms). We define the semantics of ◰ analogously to those discussed in Section 3.4.1 for the loop-lifted staircase join. In fact, the algorithm sketched there fits perfectly into our current context.

The Step Operator ◰

The step operator ◰ consumes a context relation $q(e)$ of schema $\langle iter, item\rangle$. As illustrated on the right, this relation encodes the context sequences $(\gamma_{i,1}, \ldots, \gamma_{i,s_i})$ for n iterations in a single relation. The order among context nodes within an iteration is insignificant to the location step result. Dropping the *pos* column from the loop-lifted sequence encoding thus yields a suitable input for ◰. In return, the output of operator $◰_{\alpha,\nu}$ will again be of schema $\langle iter, item\rangle$ and contain the result set for the location step $e/\alpha::\nu$ (axis α, node test ν).

iter	*item*
1	$\gamma_{1,1}$
1	$\gamma_{1,2}$
⋮	⋮
1	γ_{1,s_1}
⋮	⋮
n	$\gamma_{n,1}$
⋮	⋮
n	γ_{n,s_n}

Several implementations proposed for ◰ will automatically avoid the production of duplicates in the step result (*e.g.*, staircase join). However, we do not assume this property to be provided, but explicitly enforce duplicate elimination in terms of the algebra operator δ.

In the XQuery language specification, location steps are required to return their result in document order. In consistence with a loop-lifted sequence encoding, we

can easily enforce this order as the logical order of the expression result if we create a new column *pos*, ordered according to the node surrogates γ_i in column *item*. Hence, the algebraic subplan

$$q'(e/\alpha::\nu) \equiv \begin{array}{c} \varrho_{pos:\langle item\rangle\|iter} \\ | \\ \delta \\ | \\ \text{⌐}_{\alpha,\nu} \\ | \\ \pi_{iter,item} \\ | \\ q(e) \end{array} \tag{4.9}$$

evaluates the location step $e/\alpha::\nu$ in consistence with our loop-lifted compilation strategy. However, before we translate this plan into a generic compilation rule, we will slightly refine our notation and introduce the handling of *live nodes* first.

Live Node Sets: Multiple Tree Node Sources within a Single Query

To successfully complete its task, operator ⌐ requires access to the storage tables that contain the detailed node information referenced by surrogates γ_i in ⌐'s input $q(e)$. In Chapters 2 and 3, database table `doc` has played this role, hosting an explicitly shredded XML instance. Here, we extend this idea to allow for the processing of XPath location steps over *multiple* XML documents in a single query as well as over *transient* XML fragments constructed at runtime within the query itself.

In Section 4.1.1, we have added a column *frag* to our relational tree encoding to prepare the ground for such multi-document support. The semantics of XPath location steps demands that the evaluation of a location step may never escape the XML document/fragment that holds the step's context set, thus we can use column *frag* to make this semantics explicit to the relational step evaluation. Speaking in terms of the SQL-based XPath evaluation in Chapter 2, *e.g.*, the conjunctive predicate

$$\texttt{AND}\ \ v.frag = v'.frag$$

added to the query template $sql(e)$ would ensure the semantically correct evaluation of axes `preceding` and `following` (while mere *pre*/*size* predicates would likely lead to an escape from the valid XML tree).

However, it would be even more beneficial if we could infer information about fragments relevant to a particular expression evaluation already at compile time. Queries could then be run against a *subset* of relation `doc` only or even use a distinguished base table for each XML instance. In the presence of node constructors, the effect would be even more profound. Nodes constructed at runtime will lead to transient document containers that lack all index support. Constraining the

step evaluation to few of these containers only or—even more effective—to fully indexed base relations would significantly improve runtime performance.

Our compiler covers these aspects in terms of the *live node set* Δ, associated with the algebraic equivalent of any XQuery subexpression. We will instantiate a new live node set

(i) for any call to the XQuery *built-in function* `fn:doc()` (a compilation rule for this built-in will seed the respective base relation as a new live node set into the translation process[5]) and

(ii) for any *node construction* operator occurring in the query (the generated algebra code will produce a new, transient node container).

The inference rules for the remaining XQuery constructs will then maintain and propagate the information on live node sets throughout the compilation process and make it available to any XQuery subexpression encountered. For example, compilation rule SEQ associates the result of the sequence construction (e_1, e_2) (with $e_1 \Mapsto (q_1, \Delta_1)$ and $e_2 \Mapsto (q_2, \Delta_2)$) to the union $\Delta_1 \cup \Delta_2$ of the two live node sets involved (cf. page 81). Intuitively, node surrogates in the expression result may reference entries from either live node set.

Combining Live Node Sets. The complete compilation rule set comprises rules that *establish* new live node fragments (see above), *propagate* the live node information of their arguments, or *combine* two live node sets in terms of the union operator $\cup$. In effect, the live node set associated with each expression is the set union of multiple, say k, node containers:

$$\Delta = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_k \ . \tag{4.10}$$

Each of the Δ_i describes a single base relation in the system's persistent document storage or the result of a single element construction operator. Thus, two node containers Δ_i and Δ_j will either host *disjoint* tree fragments or encode the *same* base table/construction result:

$$\Delta_i \cap \Delta_j \neq \emptyset \quad \text{iff} \quad \Delta_i = \Delta_j \ . \tag{4.11}$$

This allows for an efficient computation of the set union in (4.10). Duplicates do *not* need to be expensively removed tuple by tuple after joining the live node sets. Rather, the system can rule out duplicates on the level of node containers

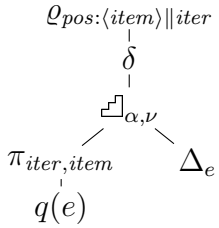
[5]The compilation rule for `fn:doc()` inherently depends on the specific document storage and we will omit its explicit statement here. The MonetDB/XQuery system implements document access as a low-level primitive added to the database kernel.

(whose number is usually small). In most cases, duplicate tree fragments may even be removed at *compile time*. The deferral into the runtime system is, in fact, only required if the built-in function `fn:doc()` is applied to a *computed* URI argument (*e.g.*, `fn:doc(fn:concat("auction",".xml"))`).

Compiling XPath Location Steps

Operator $\leftthreetimes$ represents the prototypical case where access is required to the live node set Δ. This is why we introduce Δ as a second argument to $\leftthreetimes$:

$$q(e/\alpha::\nu) \equiv$$



. (4.12)

This completes the relational plan for the location step $e/\alpha::\nu$. In the following, we will only omit the statement of Δ as an argument to $\leftthreetimes$ if the live node set information are clear from the context or irrelevant for the discussion.

From subplan 4.12, we conclude compilation rule Step as follows

$$\frac{\Gamma; \mathsf{loop} \vdash e \Rightarrow (q_e, \Delta_e)}{\Gamma; \mathsf{loop} \vdash e/\alpha::\nu \Rightarrow \left(\varrho_{pos:\langle item\rangle\|iter}\left(\delta\left((\pi_{iter,item}(q_e)) \leftthreetimes_{\alpha,\nu} \Delta_e\right)\right), \Delta_e\right)} \quad \text{(Step)}$$

XPath navigation will never escape the live node set Δ_e that hosts the context nodes in $q(e)$. The result of the step evaluation will in turn be associated to Δ_e.

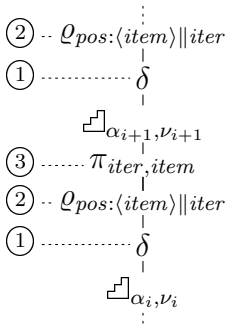
Optimization Hooks in XPath Step Evaluation

The discussion above suggests two hooks with respect to the optimization of XPath subexpressions that we will briefly sketch here.

XPath Step Bundling. It is typical for XQuery expressions to navigate through the XML tree structure in terms of multiple, say k, location steps:

$$e/\alpha_1::\nu_1/\alpha_2::\nu_2/\cdots/\alpha_k::\nu_k \ .$$

If compiled according to Rule Step, the evaluation of the above path expression will repeatedly (see also the plan excerpt on the right)

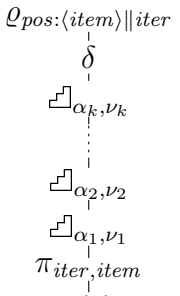


① remove *duplicates* from the result of evaluating $\text{◱}_{\alpha_i,\nu_i}$ as requested by the XPath semantics,

② establish a new *pos* column that brings the result into *document order* (in terms of the row numbering operator ϱ), and then

③ drop column *pos* to turn the result into the context *set* for the subsequent step operator $\text{◱}_{\alpha_{i+1},\nu_{i+1}}$.

Particularly the combination of bullets ② and ③ seems wasted effort in that situation. Depending on the physical order guarantees of a system's ◱ implementation, the assignment of row numbers may involve an expensive sort process of its argument. Avoiding this would surely be desirable. But also the duplicate elimination step ① may cause significant performance penalties if the successive path step does not exploit the duplicate-freeness.



Due to the definition of operator ◱—the output of any ◱ may directly be used as an input to a subsequent ◱—we can do better: extending Rule STEP to translate multi-step paths *as a whole* (see plan excerpt on the left) avoids the application of any ϱ and δ operators except the ones that establish the final path result.

Though the avoidance of these two operators is the most visible effect of bundling XPath location steps, the new operator trees may additionally trigger rewrites in the DBMS optimizer that go even further: with the entire sequence of step joins available as a whole, the optimizer may freely modify join order or—if suitable support is available to the system (*e.g.*, in terms of the TwigStack algorithm [Bruno02])—compute the entire XPath expression as a whole.

The XPath evaluation techniques in Chapters 2 and 3 may benefit from the rewrite as well. To complete its task, a region-based step evaluation must resolve each context node's surrogate γ (a preorder rank) to retrieve the remaining node properties. We may avoid this additional lookup in the document relation if the information is still at hand from a directly preceding step evaluation. In the physical execution plan, this becomes apparent through a significantly reduced number of accesses to the persistent document relation doc.

Linearity of ◱. The semantics of XPath binds the nodes processed for each context node to the fragment that is hosting the context node. In effect, for a context set e that is associated with a live node set of multiple tree fragments

$\Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, we may safely evaluate $\boxdot$ on each fragment in separation:[6]

$$\begin{array}{c} q(e) \boxdot_{\alpha,\nu} (\Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n) \\ \Leftrightarrow \\ \big(q(e) \boxdot_{\alpha,\nu} \Delta_1\big) \cup \big(q(e) \boxdot_{\alpha,\nu} \Delta_2\big) \cup \cdots \cup \big(q(e) \boxdot_{\alpha,\nu} \Delta_n\big) \quad . \end{array} \tag{4.13}$$

Rewriting step evaluation in the $\Rightarrow$ direction of this equivalence will be particularly efficient if nodes from a persistent document container are included in the context set e. Hence, if we trade

$$q(e) \boxdot_{\alpha,\nu} (\mathsf{doc}_1 \cup \mathsf{doc}_2 \cup \cdots \cup \mathsf{doc}_n \cup \Delta_{\text{trans}})$$

for the equivalent expression

$$\big(q(e) \boxdot_{\alpha,\nu} \mathsf{doc}_1\big) \cup \big(q(e) \boxdot_{\alpha,\nu} \mathsf{doc}_2\big) \cup \cdots \cup \big(q(e) \boxdot_{\alpha,\nu} \mathsf{doc}_n\big) \cup \big(q(e) \boxdot_{\alpha,\nu} \Delta_{\text{trans}}\big)$$

(where doc_i denotes persistent document relations and Δ_{trans} summarizes all transient fragments constructed at runtime), the system will perform the bulk of the work (usually $|\Delta_{\text{trans}}| \ll |\mathsf{doc}|$) on indexed base tables. In contrast, the union operators in the original query prevented the use of base table indexes outright.

Property (4.13) coincides with the definition of *linearity* by Gluche *et al.* [Gluche97]. The exploitation of the linearity property in OQL view definitions allows for an efficient maintenance of materialized views there. The same observations may open the door for compiler optimizations that minimize those parts of a query that need to operate on transient live nodes.

4.4.2 Element Construction

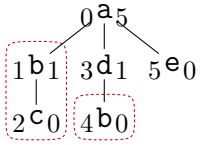
An important aspect of the XQuery language is its capability to group and restructure the XML trees retrieved during query processing. For this purpose, the language has been equipped with node construction facilities that allow for the construction of transient tree nodes on the fly. The element constructor

$$\texttt{element \{ } e_1 \texttt{ \} \{ } e_2 \texttt{ \} } \ ,$$

for example, instantiates a new element using the content expression e_2 and the tag name e_1. More specifically, given an expression e_1 that evaluates to a qualified XML tag name t and a content sequence $e_2 = (v_1, v_2, \ldots, v_k)$ consisting exclusively of nodes, *i.e.*, e_2 is an instance of `node()*`, the element constructor

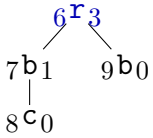
① creates a new element node r with tag name t and then

[6] We have omitted the projection step $\pi_{iter,item}$ on $q(e)$ here for ease of readability.



iter	pos	item
1	1	1
1	2	4

(a) Old tree fragment with content expression $q(\$v)$.



iter	pos	item
1	1	6

(b) New tree fragment and element construction result.

	pre	size	level	kind	prop	frag
	0	5	0	elem	a	0
	1	1	1	elem	b	0
	2	0	2	elem	c	0
	3	1	1	elem	d	0
	4	0	2	elem	b	0
	5	0	1	elem	e	0

old tree fragment

	pre	size	level	kind	prop	frag
①	6	3	0	elem	r	1
	7	1	1	elem	b	1
	8	0	2	elem	c	1
②	9	0	1	elem	b	1

element constr. result

(c) Relational encodings of old and new tree fragments (*pre*/*size*/*level* encoding).

Figure 4.8: Element construction and resulting tree fragment. The range encoding (*pre*/*size*/*level*) reduces the renumbering overhead to copy subtrees.

② copies the k subtrees rooted at the nodes v_i and arranges the copies under the common root r.

The new node r is then returned as the overall expression result. Note that both steps generate nodes with a new, unique *node identity*, disjoint from both construction arguments. Figure 4.8 uses the range encoding scheme (*pre*/*size*/*level*) to give an example of the evaluation of the element constructor

```
let $v := e/descendant::b return
  element { "r" } { $v } ,
```

for which we assume that e evaluates to the singleton sequence containing the root node `a` of the tree depicted in Figure 4.8(a). After evaluating the path `e/descendant::b`, variable $\$v$ will be bound to the sequence containing the two element nodes with tag name `b` (preorder ranks 1 and 4, see Figure 4.8(a) for its loop-lifted representation). Figure 4.8(b) shows the expected outcome of the expression: the copies of the subtrees rooted at the two `b` now share the newly constructed root node `r`.

Figure 4.8(c) illustrates how element construction is reflected in the associated live node sets. The establishment of the new tree fragment turns out to be particularly easy to implement with an XML representation based on *pre*/*size*/*level*:

① A new root node `r` is created for the new tree fragment and assigned the next available preorder rank (6 in our case).

② Nodes in the affected subtrees are then appended to the new tree fragment with their *size*, *kind*, and *prop* properties unchanged. Properties *level* and *pre* shift by a constant offset only; a system that infers preorder ranks from the physical tuple order may not need to explicitly store the latter at all (cf. MonetDB's `void` column type [Boncz02]).

This procedure is easily expressible in an algebraic manner. In [Grust04c], *e.g.*, we sketch an implementation of the same idea in SQL.

A Compilation Rule for `element { }`

Naturally, the required steps to implement element construction depend on the underlying XML document representation. In order to formulate a compilation rule for the task, we encapsulate these specifics into the operators ε (element constructor), τ (text node constructor), etc. of our relational algebra. Given

(i) the encoding of the tag name e_1 for each iteration (schema $\langle iter, pos, item\rangle$),

(ii) the content expression e_2 in its loop-lifted representation, and

(iii) the live node set Δ_2 associated with e_2,

operator ε, *e.g.*, evaluates the expression `element { `e_1` } { `e_2` }` and returns the pair (q_e, Δ_e), consisting of q_e, the relational encoding of the element construction result, and Δ_e, its associated live node fragment. With this definition of ε, the derivation of Rule Elem becomes straightforward:

$$\frac{\Gamma; \mathsf{loop} \vdash e_1 \Rightarrow (q_1, \emptyset) \qquad \Gamma; \mathsf{loop} \vdash e_2 \Rightarrow (q_2, \Delta_2) \qquad (q_e, \Delta_e) \equiv \varepsilon(q_1, q_2, \Delta_2)}{\Gamma; \mathsf{loop} \vdash \texttt{element \{}\, e_1 \,\texttt{\}\ \{}\, e_2 \,\texttt{\}} \Rightarrow (q_e, \Delta_e)}\ . \qquad \text{(Elem)}$$

Observe that subexpression e_1 evaluates to a tag name and therefore is not associated with any live node fragment. Only newly constructed nodes can contribute to the overall expression result and we record the new tree fragment Δ_e as q_e's live node set.

4.4.3 A Note on Side-Effects

Besides the challenges of their efficient implementation, the node construction facilities bear some semantical peculiarities within the XQuery language. The establishment of new node identities during the construction process introduces a *side-effect* that disrupts the language's *referential transparency*. To illustrate, the query

$$\texttt{let \$}v\texttt{ := <a/> return \$}v\texttt{ is \$}v \qquad (Q_6)$$

(evaluating to `true()`) is *not* equivalent to the "unfolded" expression

```
<a/> is <a/> .
```

obtained by replacing all occurrences of $\$v$ by its binding expression `<a/>`. While the former expression constructs a single element node and binds variable $\$v$ to it, the latter produces two distinct element instances and, hence, returns `false()` as its result.

Side-Effects in Relational Query Plans

For a sound evaluation of arbitrary XQuery expressions, we obviously must make our relational implementation aware of this semantical difference. Two different approaches have proven their practicability in actual system setups:

Stateful XQuery Compilation. In the first place, the rule set we have described leads to a compilation of XQuery expressions *bottom-up*. However, inspired by the monad-based information passing in functional programming languages [Wadler90], we may just as well use it to propagate additional *state information* through the compilation process. This state information is then made available to the ε operator which in turn is responsible for the reproduction of identical surrogates for nodes of the same identity.

This idea has resulted in the construction of an SQL-based XQuery evaluator, for which we refer the reader to [Grust04c] for details.

DAG Representation for Algebraic Plans. MonetDB/XQuery, the implementation accompanying this thesis, follows a different approach to implement node construction in a semantically correct way. The Pathfinder compiler exploits the significant amount of *identical subplans* within the algebraic plan trees and thus records them as *directed acyclic graphs (DAGs)* instead.

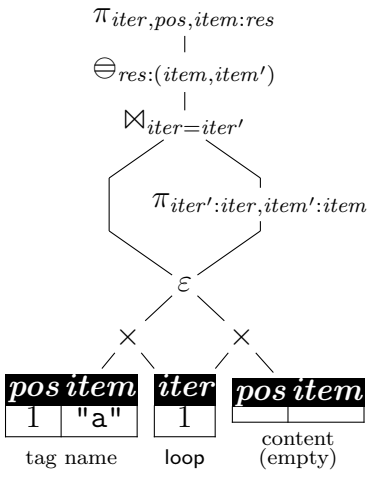


Figure 4.9: Plan DAG for Q_6.

This way, node construction operators ε, τ, ... must be listed only once for each instance of the respective functionality in the XQuery expression text. The above query Q_6, *e.g.*, is compiled into the plan DAG shown in Figure 4.9 on the right by the Pathfinder compiler. Besides the two references to variable $\$v$, this DAG contains the construction of element `<a/>` only once, as desired. We will learn more about Pathfinder's DAG representation and optimization later in Chapter 5.

4.5 Support for Dynamic Type Tests

Finally, we want to assess how to provide efficient support for XQuery's type testing facilities on a relational execution engine. The `typeswitch` clause in particular allows for querying the *dynamic type* of any XQuery subexpression at runtime. The expression

$$
\begin{array}{l}
\texttt{typeswitch (}e\texttt{)} \\
\quad \texttt{case } \tau_1 \texttt{ return } e_1 \\
\quad \texttt{case } \tau_2 \texttt{ return } e_2 \\
\quad\quad \vdots \\
\quad \texttt{case } \tau_n \texttt{ return } e_n \\
\quad \texttt{default return } e_{\text{def}}
\end{array}
\qquad (Q_7)
$$

will successively test expression e for a subtype relationship with all types τ_i. For the first test that succeeds, the respective expression e_i is evaluated and its result is returned as the overall query result. If no match can be found, the outcome is determined by the `default` expression e_{def}.

4.5.1 XQuery Subtype Semantics

With XML trees as the underlying data model, the evaluation of the `typeswitch` expression Q_7 involves the comparison of *tree-structured* data types. The task of resolving the *structural* subtype relation $\tau_1 <: \tau_2$ (with types τ_1 and τ_2) is usually performed as an *inclusion test* of their corresponding tree automata (see, *e.g.*, [Hosoya05]) or by *reducing* their corresponding *regular expressions* until the relation can be decided trivially [Kempa03, Antimirov95]. The computational complexity of these tests, however, rules out their actual application to high-volume XML processing. Taking additional constraints of the XQuery surface language into account, though, it turns out that we can reduce the subtype test to simple relational algebra primitives, efficiently supported by existing back-ends.

The key to this approach is the restriction of the types τ_i in Q_7 to *sequence types* [Boag05]. In a nutshell, sequence types consist of two components: an *item type* t that describes the type of each item in the sequence and a *cardinality* $c \in \{\texttt{1}, \texttt{?}, \texttt{+}, \texttt{*}\}$. The subtype relationship of sequence types may be tested for both components *in separation*, which the XQuery Formal Semantics also refer to as subtype *matching* [Draper05, § 8.3.1]. Besides the evaluation of these two components, no further look into the structure of τ_i or the type τ_e of expression e is required to decide whether e is an instance of τ_i (*i.e.*, $\tau_e <: \tau_i$).

Each item type may only be derived from a single base type, either by *restriction* (defined in terms of an `xs:restriction` element in XML Schema) or *extension* (`xs:extension` in XML Schema, respectively) and the subtyping judgment $<:_{item}$

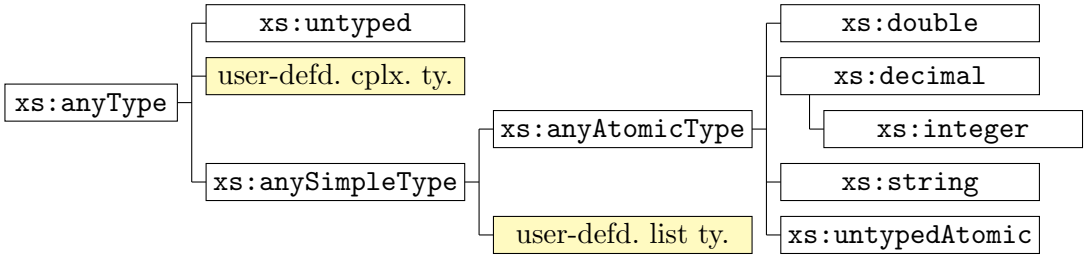


Figure 4.10: XQuery type hierarchy (excerpt) [Fernández05].

for XQuery item types strictly follows the derivation hierarchy in an XML Schema specification. We can easily build up a derivation tree that describes this type hierarchy T at query compilation time. In Figure 4.10, we list T for an excerpt of XQuery's built-in item types. User-defined types in XML Schema documents[7] will contribute to the hierarchy tree in a straightforward fashion.

4.5.2 Sequence Type Matching on Relational Back-Ends

The resulting hierarchy tree is a data structure that we know well how to handle efficiently by relational means: in Chapter 2 we discussed the encoding of (XML) trees in terms of each node's pre- and postorder ranks in great detail. Here, we will use the same idea to elegantly back up type matching in our relational setup.

Pre- and Postorder Ranks for XQuery Item Types

To make user-defined types available to the query processor, their XML Schema definitions must explicitly be referenced in the respective XQuery prolog. This makes the entire type hierarchy T available to the system *at query compilation time* and we annotate each *type* t with its pre- and postorder ranks $pre_T(t)$ and $post_T(t)$ to prepare the compilation of relational sequence type matching (note that we are assigning ranks to *types* here, not XML tree nodes).

Now, if we have the pre_T and $post_T$ values available for each item type t_i of sequence item x_i in $e = (x_1, x_2, \ldots, x_k)$, we can express $<:_{item}$ using ancestor/descendant relationships in T:

$$\begin{gathered} \tau_e <:_{item} t \\ \Leftrightarrow \\ \forall i \in \{1, \ldots, k\} : pre_T(t_i) \geq pre_T(t) \ \wedge \ post_T(t_i) \leq post_T(t) \ , \end{gathered} \tag{4.14}$$

i.e., "all item types t_i in τ_e are a descendant of type t in the hierarchy T."

[7] The XQuery directive `import schema` instructs the query compiler to load additional type information from an XML Schema document.

To test this relationship, we might as well pick the minimum preorder rank $pre_T(t_i)$ and the maximum postorder rank $post_T(t_i)$ found for the items in e first:

$$\begin{array}{c} \tau_e <:_{item} t \\ \Leftrightarrow \\ \min_{i\in\{1,\ldots,k\}} \Big(pre_T(t_i)\Big) \geq pre_T(t) \ \wedge \ \max_{i\in\{1,\ldots,k\}} \Big(post_T(t_i)\Big) \leq post_T(t) \ . \end{array} \tag{4.15}$$

The computation of these aggregates is an easy task for a relational query engine and we will use the grouping operator $\mathsf{grp}_{a:\mathsf{min/max}\, b \| p}$ to describe them in our algebra.

Aggregates for Cardinality Tests

The use of aggregates blends equally well with the test of the cardinality constraint c imposed by the sequence type τ. We can test $\tau_e <:_{card} c$ by simply *counting* the items in e:

$$\tau_e <:_{card} c \quad \Leftrightarrow \quad \begin{cases} \text{count}\,(e) = 1 & \text{if } c = \mathsf{1} \\ \text{count}\,(e) \leq 1 & \text{if } c = \mathsf{?} \\ \text{count}\,(e) \geq 1 & \text{if } c = \mathsf{+} \\ \mathsf{true} & \text{if } c = \mathsf{*} \end{cases} . \tag{4.16}$$

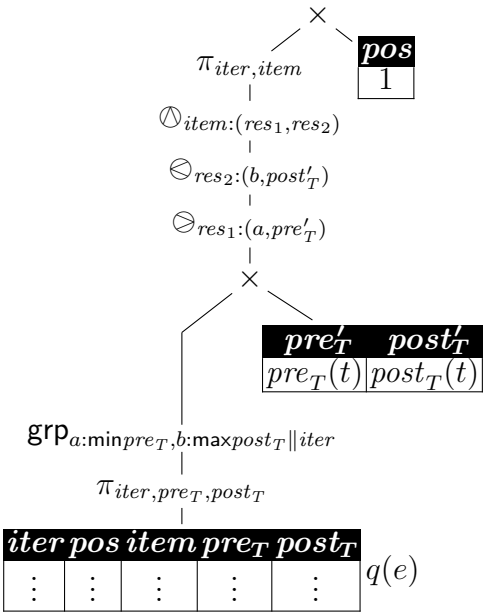
This translates the cardinality test into another aggregation over the input expression e.

Item Types for Loop-Lifted Sequences

Testing the item type relationship using Equation 4.15 in our relational setup requires the explicit availability of properties $pre_T(t_i)$ and $post_T(t_i)$ for each sequence item x_i. If we add both properties to the relational encoding of e, Equation 4.15

directly translates into a relational plan:

$$q(\tau_e <:_{item} t) \equiv$$



. (4.17)

The plan performs the required grouping for properties $pre_T(t_i)$ and $post_T(t_i)$ in a single step and for all sequences in all iterations (operator grp). The remaining operators compare the aggregates to their counterparts in item type t, then reassemble the usual $\langle iter, pos, item\rangle$ column schema.

Note that the assumption about the availability of pre_T and $post_T$ blends well with the semantics of XQuery. XML tree nodes, *e.g.*, are generally assumed to be *untyped* in XQuery. Only the explicit *validation* of a node v makes further type information available to query processing. In relational terms: the relational code for the XQuery primitive `validate { v }` will *annotate* the sequence encoding of v with new columns pre_T and $post_T$. An efficient means to implement this procedure has been suggested, *e.g.*, by Grust and Klinger [Grust04b].

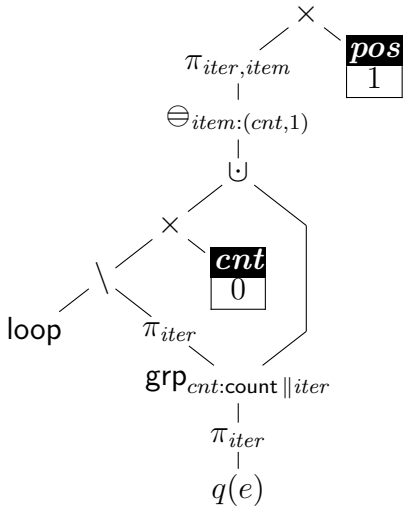
To retain the compositionality of our translation, the compiler will use the extended $\langle iter, pos, item, pre_T, post_T\rangle$ schema throughout the compilation process. For non-validated expressions, columns pre_T and $post_T$ will be filled up with suitable constants as needed (*e.g.*, `xs:untyped` or `xs:untypedAtomic` for XML tree nodes).

Testing Cardinalities

We can handle the second component of XQuery sequence types in quite a similar fashion. Equation 4.16 suggests the use of the aggregate count(e) to test e's conformance to the sequence type cardinality c. However, an important aspect of our sequence encoding takes its toll here: if an expression e evaluates to the

empty sequence `()` in an iteration i, this fact will be represented in the loop-lifted encoding as the *absence* of tuples with $iter = i$. Consequently, iterations with $\text{count}(e) = 0$ will be missing in the aggregation result if $\tau_e <:_{card} c$ is evaluated as prescribed by Equation 4.16.

The information required to "fix" this problem is available in the form of the loop relation that lists all values of *iter* valid in the current iteration scope. The following plan, *e.g.*, implements the test $\tau_e <:_{card} \texttt{1}$, adding $cnt = 0$ to the aggregation result for all iterations in e that evaluate to the empty sequence. Note how the left branch of the disjoint union contributes those iterations that are in loop, but not in the output of grp:

$q(\tau_e <:_{card} \texttt{1}) \equiv$ 

. (4.18)

This idea of using loop to correctly handle the empty sequence `()` directly leads to a compilation rule for the XQuery built-in function `fn:count ()`, but also for other XQuery aggregates (*e.g.*, `fn:sum ()`, `fn:empty ()`, ...).

Putting Things Together

The value of an XQuery expression e *matches* a sequence type τ if it satisfies τ's constraints on both, τ's item type t ($\tau_e <:_{item} t$) and the sequence cardinality c imposed by τ ($\tau_e <:_{card} c$). The conjunction of both judgments will, hence, make a valid translation for XQuery subtype matching. An actual implementation will usually wrap both aspects into a single subplan. The involved aggregates may then, *e.g.*, be determined by a single grp invocation, possibly saving a considerable amount of computation.

We have not yet touched upon the optional restriction of node types to a given tag name. An XQuery sequence $e = \texttt{(}x_1\texttt{, ... ,}x_k\texttt{)}$, *e.g.*, matches the sequence type `element (`n`,` t`)`c only if

(i) the type annotation of all items x_i matches the item type t ($\tau_e <:_{item} t$),

(ii) the number of items k in e is consistent with the cardinality c, and

(iii) all items x_i in the sequence are element nodes with tag name n.

The third test on tag names seamlessly integrates into the implementation of sequence type matching if we define the tailor-made aggregation function tag:

$$\underset{i\in\{1,\ldots,k\}}{\text{tag}}(x_i) = \begin{cases} n & \text{if } \forall i : prop(x_i) = n \\ * & \text{otherwise} \end{cases} .$$

This translates the constraint on tag name n into the test $\text{tag}(e) = n$, which can be easily added to a relational plan. The outcome is a type matching procedure that handles all sequence type instances allowed by the XQuery language specification in terms of simple value aggregates. Such aggregates are efficiently implemented in most of the existing database systems.

4.6 XQuery on DB2

Loop-lifted XQuery compilation generates purely relational execution plans for XQuery expressions of arbitrary nesting. This way, *any* relational database processor can serve as an efficient host to XQuery. To back up this claim, we translated the algebraic operators in Table 4.1 into SQL and ran a number of queries from the XMark benchmark set on the DB2 instance from Chapter 2.

4.6.1 A Loop-Lifted XQuery-to-SQL Translation

The relational algebra listed in Table 4.1 is sufficiently simple to be implemented on top of an SQL system. In [Grust04c], we devised a compilation procedure that expresses the concept of loop-lifting entirely in SQL. We used exactly the same translation procedure to pursue the studies on DB2 that we will describe in the following. The rule set of this XQuery-to-SQL compiler suffices, *e.g.*, to compile all queries from the well-known XMark benchmark [Schmidt02] into their SQL counterpart. We chose a number of them for further investigation on DB2.

As in Chapter 2, we let the system determine the appropriate index support completely on its own. Before running our experiments, we provided the DB2 index advisor `db2advis` with a workload of all investigated queries and created indexes as suggested by the advisor. Suggestions included those indexes that we already found beneficial in Chapter 2. No further "wizardry" was applied to tune our database server.

Despite the quite unusual style of SQL code that resulted from our compilation process, DB2 did remarkably well in optimizing our queries. As a consequence of their origin in XQuery, the generated plans exhibit a number of interesting optimization opportunities. To assess their potential, we were able to make some of these opportunities explicit in the SQL code shipped to our DB2 instance. Several more advanced optimization hooks, however, require explicit support from the relational back-end. In the next chapter, we will thus look into the MonetDB/XQuery system as a representative for such a system.

4.6.2 XPath Bundling and Use of OLAP Functionality

The output of our XQuery-to-SQL compiler can reach considerable size. If translated in strict accordance to the SQL compilation procedure listed in [Grust04c], query *Q1* from the XMark benchmark, *e.g.*, results in an operator tree of 491 nodes as reported by DB2's explain utility `db2expln`. The style of the generated plans is quite different to the usual $\pi{-}\sigma{-}\bowtie$ pattern, posing a significant challenge to any relational query optimizer.

XPath Location Step Bundling

In case of the same XMark query *Q1*, the DB2 optimizer seemed to particularly miss the chance to *bundle adjacent XPath location steps*. The seven steps in XMark query *Q1* led to no less than 115 accesses to the base relation `doc` for the loop-lifted query.

We already discussed the effectiveness of bundled XPath evaluation in Section 4.4.1. Such a bundling is easily expressible on the level of SQL and, in fact, it cut down the number of accesses to the document container `doc` by almost a factor of six if applied to the SQL code of Query *Q1*. The respective figures are lined up in Table 4.2.

Exploiting OLAP Functionalities

To implement XQuery's tight constraints on document, sequence, and iteration order, loop-lifted query plans make heavy use of the row numbering operator ϱ to impose logical order on the unordered back-end data model. A possible means to express this operator in SQL is the use of functionalities from the SQL:1999 *OLAP amendment* [Melton03], the `ROW_NUMBER()` function in particular (Section 4.1.2). The ranking and partitioning functionalities provided by this operator make quite an ideal match for our choice of sequence encoding and representation of iteration, *i.e.*, a *single* relation encodes the sequence value for *all* iterations of a `for`-loop.

Optimization	execution time [s] 114 KB	293 KB	1.1 MB	3.3 MB	# doc accesses
no optimization	2,132	12 hours	—	—	115
bundling XPath steps	0.003	0.006	0.106	0.566	20
use of OLAP functionality	0.030	0.219	1.03	9.18	13
OLAP and bundled XPath	0.002	0.002	0.002	0.004	11

Table 4.2: Effectiveness of optimizations for XMark query *Q1*. The use of DB2's OLAP functionalities and the bundling of XPath location steps (Section 4.4.1) cut down accesses to base relation doc quite significantly and reduced execution times by orders of magnitude.

On the other hand, the availability of OLAP extensions is not a strict requirement for SQL-based XQuery evaluation. In fact, the SQL-based loop-lifting compiler we described in [Grust04c] does not rely on the availability of OLAP functionality at all. However, Table 4.2 shows that their use has quite a significant effect on query performance in comparison to the non-OLAP equivalent (labeled "no optimization"). In the case of XMark query *Q1*, the application of `ROW_NUMBER()` to implement ϱ accelerates query execution by orders of magnitude. DB2's query optimizer readily accepts the explicit statement of our renumbering intents, observable in a significant speedup in query execution time.

Both optimizations together, the bundling of XPath location steps and the exploitation of OLAP functionalities, can speed up SQL-based XQuery execution by several orders of magnitude. To demonstrate, we have listed the query execution times for small document instances in Table 4.2 with and without both optimizations applied. Given the obvious benefit of the two, all remaining experiments include both optimizations.

4.6.3 Live Node Sets: Compile-Time Information for Accelerated Query Evaluation

The precise information about the base tables or transient node containers that host intermediate node sequences can have a significant impact on the choice of efficient access methods to the respective node properties. A disk-based database system will almost certainly try to access persistent base tables with the help of efficient indexes. Their use becomes infeasible, however, if the system is to access *computed* node containers.

On the other hand, a system needs to ensure the consistent handling of the *node construction* facilities in XQuery, the uniqueness of node identity in particular.

A possible means to guarantee this consistency is by simply collecting all XML trees and fragments encountered during the compilation in a *single* document container. Essentially, this container collects all side effects that occur during element construction.

A similar approach has been followed, *e.g.*, in the relational XQuery mapping described by DeHaan *et al.* [DeHaan03], which actually collects sequences *and* constructed nodes in a single relational view. The resulting *union* of multiple intermediate results, however, renders the resulting document encoding inaccessible to persistent index structures, a price that becomes increasingly unaffordable if the underlying repository is growing.

It is exactly this situation in which the execution engine can profit from the tracking of *live node set* information in our compiler. The effect becomes directly observable, *e.g.*, for Query *Q13* from the XMark benchmark:[8]

```
for $i in fn:doc("auction.xml")/site/regions/australia/item
  return
    element item { (element name { $i/name/text() },
                    $i/description) }
```
(*Q13*)

To evaluate this query, the system eventually creates the `name` element nodes, containing subtree copies of the nodes returned by `$i/name/text()`. Subsequently, if it is to compute the path `$i/description`, the system may either

① base its evaluation on the *entire* set of nodes encountered so far, *i.e.*, on the union containing the persistent `doc` table and the fragment Δ obtained when constructing the `name` elements (similar to the approach of [DeHaan03]) or

② exploit the compiler's *live node set* information and evaluate the step on the persistent `doc` relation only.

Likewise, to compute the subtree copies for the second element constructor (element `item`), the execution engine may optionally

③ benefit from the *linearity* of ⏍ (which we use to implement subtree copying) and perform subtree copying on the two live node set components `doc` and Δ in separation (cf. page 97).

Each of the evaluation strategies ② and ③ further increases the use of efficient *indexed* access to the persistent document relation `doc`. The effect is a significant improvement in query performance, growing with the size of the respective XML document instance. Table 4.3 lists the actual execution times we measured for different XML instance sizes.

[8] In the original XMark *Q13* query, the inner constructor creates an attribute node. Our discussion is not affected by this adaptation.

Optimization	execution time [s] 114 KB	1.1 MB	3.3 MB	11 MB
① evaluation on document union	0.032	2.83	1,139	6,338
② live node set exploitation	0.027	0.486	24.1	5,971
③ linearity of ⏍	0.014	0.126	0.361	1.32

Table 4.3: Exploiting live node set information and the linearity of ⏍ significantly reduces the cost to access node containers in XMark *Q13*.

4.6.4 XMark on DB2

To summarize and assess the overall performance of loop-lifted XQuery evaluation on DB2, we selected a number of queries from the XMark benchmark. After compiling them into SQL by hand, we ran each of them on XML instances of various sizes, with the total execution times documented in Figure 4.11.

Our test set focuses on the XQuery constructs we have discussed in the foregoing sections. All five queries contain XQuery `FLWOR` clauses to iterate over intermediate results and path expressions of various lengths to access nodes from the document. Query *Q1* is quite centered around XPath evaluation, while Queries *Q2* and *Q13* additionally employ element constructors to establish their result. Queries *Q6* and *Q7* both exhibit path steps with recursive semantics and make use of the aggregation function `fn:count()`.

All queries show a linear scaling over the entire range of document sizes. Execution times stay reasonable even for the 1.1 GB XML instance, where the system would still allow for interactive querying.

Effective Path Rewrites for Query *Q1*. Benchmark query *Q1* is essentially a measure for the system's XPath performance for the path

```
/site/people/person[@id = "person0"] .
```

We have already found DB2 to be an efficient XPath processor for encoded XML data. The outcome of Query *Q1*, however, is still remarkable. Despite the complexity of the resulting query plan (45 operator nodes in total), DB2 actually detected the chance to evaluate this path in a *backward* fashion. Starting from the highly selective predicate on the `@id` attribute (accessed via an index on attribute values), the system processes the above path in a backward fashion from leaf to root. DB2 figured out the superiority of this strategy for Query *Q1* by purely relational means. The same decision had turned out to be quite a challenge to optimizers of native XML databases in the past [McHugh99].

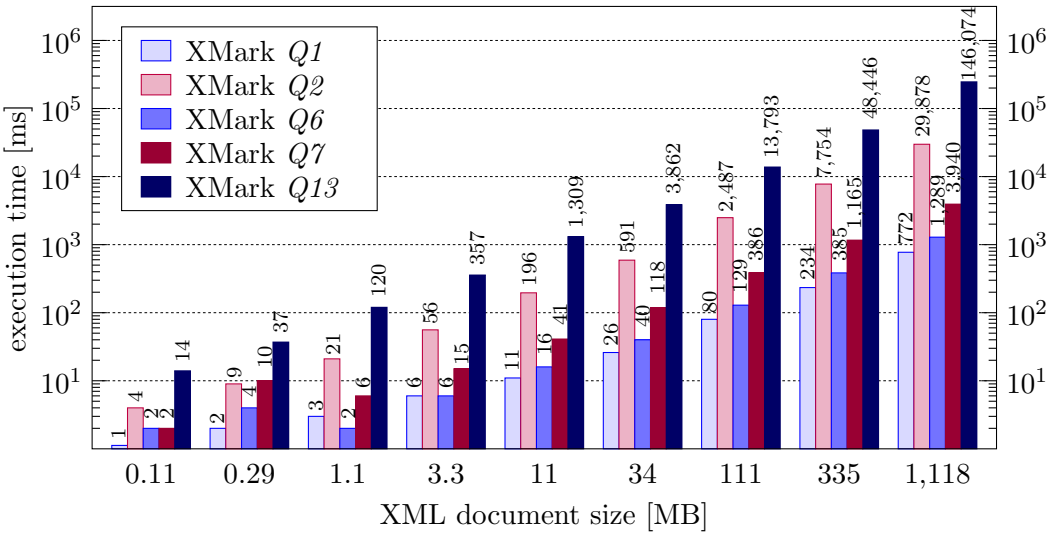


Figure 4.11: XQuery on DB2: the XQuery-to-SQL translation using our loop-lifting strategy results in an acceptable query performance even on the "innocent" SQL database.

SQL Support for Aggregate Functions. Queries *Q6* and *Q7* contain the XQuery built-in function `fn:count()` with the primary purpose to aggregate a large number of result nodes from the XML document tree. As such, they can directly benefit from the efficient implementation of aggregates in the relational system and we see interactive response times over the whole size range for both queries. But, again, we also benefit from accelerated XPath performance. Both queries make use of `descendant-or-self` XPath steps, a setup which our relational document encoding was particularly designed for.

4.7 Wrap-Up

We have already found relational database systems capable of serving as an XML mass storage and acting as efficient processors for XQuery's navigational sublanguage XPath. In this chapter, we added the missing pieces to embrace other core XQuery functionalities in a purely relational fashion. This extended our relational XML processing stack to full compliance with the XQuery language.

Three aspects of our work are essential for the implementation of a full XQuery system and make our compiler outstand among related work in the field:

(i) The compilation procedure we describe is *fully compositional*, an invaluable property if support for XQuery's arbitrary expression nesting is requested.

(ii) We do not depend on the presence of XML Schema information or DTD information (our compiler is *schema-oblivious* in the sense of [Krishnamurthy03]).

(iii) Our approach is *truly relational*. Unlike earlier work, we do not depend on specific kernel extensions to make the approach perform well (although we may benefit from them, *e.g.*, in terms of staircase join).

4.7.1 Related Research

Given the large body of research work on relational XPath evaluation and the positive performance results obtained therein, it seems quite surprising that only few approaches have been published that succeeded in leveraging this performance into the domain of XQuery. This observation is backed by the survey paper on XML-to-SQL query translation by Krishnamurthy *et al.* [Krishnamurthy03]. In fact, none of the approaches we are aware of is able to provide all the essentials listed above.

SQL Generation in the Agora System

An early attempt to consistently translate XQuery expressions into SQL has been published by Manolescu *et al.* [Manolescu01]. The *Agora data integration system* maps XQuery expressions to SQL queries over a relational tree encoding. The relational schema in use is an implementation of the edge mapping idea [Florescu99], hence, this approach is schema-oblivious like ours.

Agora's compilation procedure, however, remains limited to only a small subset of the XQuery language. In fact, the authors think that it is *impossible* to correctly translate nested XQuery expressions into single SQL queries.[9] This is mainly due to the lack of an explicit representation of *order* in their relational sequence encoding. In contrast, due to loop-lifting which explicitly accounts for *sequence* and *iteration order*, our compiler does not suffer from the same problem.

DeHaan *et al.*: A Translation Based on Dynamic Interval Encoding

The work by DeHaan *et al.* [DeHaan03] comes closest to what we have developed here. Based on the *dynamic interval encoding*, an XML tree encoding that resembles our range encoding scheme, the authors describe a compositional translation from a subset of XQuery Core into a set of SQL view definitions. Quite similar to our loop-lifted sequence representation, this translation assumes a value encoding that lists the results of parallel loop evaluations in a single relation.

[9] *"Note that in XQuery, order can appear at any level of nesting [...] therefore, correctly translating a nested order-conscious XQuery query by a single SQL query is impossible."* [Manolescu01]

Notwithstanding, the translation scheme lacks a clear distinction of the different notions of *order* in the XQuery language. Since, for example, sequence and document representations are mixed in a single relational view, the approach is unable to separate *sequence order* from *document order*. Both order notions are kept completely disjoint in our compiler, yet seamlessly integrate if sequence order is explicitly derived from document order, *e.g.*, after the evaluation of an XPath location step.

We feel that the most important drawback of the work in [DeHaan03], however, is the complexity and execution cost of the generated SQL view definitions. The compilation of path expressions, for example, leads to nested, *correlated* queries, a shortcoming that has also been identified by [Krishnamurthy03]. To evaluate these queries, the RDBMS falls back to nested loops plans, which renders the relational back-end a poor XQuery runtime environment. To achieve acceptable performance, DeHaan *et al.* indeed propose modifications to the relational engine that are specifically geared to support their dynamic interval encoding.

The XQuery Implementation in Microsoft SQL Server 2005

Modifications of the underlying engine are also essential to the relational XQuery implementation in the latest release of Microsoft SQL Server™. To accelerate the evaluation of the system's XQuery functionalities, each `XML`-typed database column may optionally be paired with a *primary XML index*. This index is an implementation of the ORDPATH tree encoding [O'Neil04] and, hence, provides for schema-oblivious relational XML storage.

The compilation of XQuery expressions into relational execution plans on the basis of this index depends on specific XML extension operators provided by the execution engine of SQL Server [Pal05]. Most notable among these operators are the system's XPath implementation *XmlOp_Path* and a direct equivalent to XQuery `for` constructs in terms of the *XmlOp_Apply* operator.

While the introduction of *XmlOp_Path* mainly attributes to performance and can be reversed without affecting the plans' semantics (much like the addition of staircase join in our setup), the *XmlOp_Apply* operator is indispensable to the semantically sound evaluation of XQuery `FLWOR` expressions in SQL Server 2005. Given a binding sequence e_1, the name of a variable v, and a `return` expression e_2, operator *XmlOp_Apply* binds v to the items in e_1 and successively evaluates e_2. As such, the operator *interrupts* the purely set-oriented execution process and falls back to an *iterated* evaluation in the presence of XQuery `for` clauses.

In this way, SQL Server does not represent a truly relational XQuery processor in the sense described here. Tailor-made kernel modifications are likely to break the bulk-oriented execution paradigm of modern database systems, which proved to be one of the essences of effective query optimization.

On the other hand, the implementation of Pal *et al.* is the only work we are aware of that succeeded in hosting XQuery on a relational system both, *efficiently* and in a *semantically correct* manner. Furthermore, SQL Server 2005 provides a seamless integration with XML Schema documents [Pal06]. The implementation of loop-lifted XQuery evaluation could surely benefit from a similar use of schema information for relational XQuery optimization.

Expressiveness of Node Construction

The *node construction* facilities in XQuery are primarily meant as a means to elegantly re-format a query's result set into suitable XML fragments. As Le Page *et al.* have found out, however, these operators also contribute to the *expressiveness* of the XQuery language [Page05]. Though our compilation procedure does not have any problems supporting this expressiveness, we saw for XMark query *Q13* that performance can significantly degrade if computed XML fragments are queried with XPath. Using live node set information, our compiler keeps accesses to such computed fragments minimal, the problem per se, however, remains.

On the other hand, there are instances in which a query could be expressed equivalently without the use of node constructors at all. This is where our compiler could benefit from the work by Le Page *et al.* The rewrite technique in [Page05] turns each node-constructing expression whose XML result contains only original nodes (a "node-conservative" expression) into an equivalent expression that does not construct new nodes. Current work is in progress in the Pathfinder project to actually perform equivalent rewrites on the basis of our compiler's relational algebra output.

4.7.2 Outlook & Perspective

An encoding of XQuery item sequences formed the key to the work we have described in this chapter: the *loop-lifted* encoding represents the items in an expression result e for *all* iterations it appears in in a *single* relation. The explicit representation of *sequence* and *iteration order* makes this encoding flexible enough to capture a significant fragment of the XQuery language by purely relational means.

Loop-Lifting in Other Domains

This flexibility suggests the use of loop-lifting in applications outside the domain of XQuery as well. An instance that is increasingly gaining attention in the software development community is the LINQ ("language integrated query") project in the Microsoft .NET Framework [Meijer05]. As a universal means to query XML as

well as relational sources within the C# language, LINQ is equipped with iteration primitives much like XQuery.

To achieve maximum performance, the C# compiler strives to delegate most of the operations expressed in LINQ to its database back-end(s), where skillful optimizers can rewrite the user input efficiently. The latest LINQ prototype provided by Microsoft does quite well in this respect.[10] Lacking fully compositional means to handle iteration, however, we found the compiler frequently "escape" to an evaluation in the C# runtime ("common language infrastructure", CLI). This is where the loop-lifting idea could step in: providing a generic model for iteration by relational means, the techniques described in this chapter could lead to a fully compositional implementation of LINQ.

Unsupported XQuery Features

The discussion in this chapter omitted several XQuery Core constructs that may nevertheless seamlessly be mapped to loop-lifted equivalents. In fact, the XQuery compiler Pathfinder embraces most of these features in a strictly loop-lifted implementation. However, we found a few XQuery Core functionalities that cannot be straightforwardly mapped into loop-lifted plan equivalents.

The Built-In Function `op:to()`. This function, available at the surface language in terms of the binary operator `to`, generates consecutive integer numbers within a given value range. Such a generation of "artificial" values is against the nature of the underlying database kernel. For an appropriate support of `op:to()`, we expect to require explicit support from the DBMS back-end.

Recursive User-Defined Functions. Effectively, we handle user-defined functions by expanding the function body into any reference to the function. This is similar to the approach taken by Manolescu *et al.* [Manolescu01]. The allowance of *recursion* in the XQuery language obviously prohibits this evaluation strategy for recursively defined functions.

Past research on *deductive databases* has brought up a large body of literature covering the efficient handling of recursive queries (see [Ramakrishnan95] for a survey) and support for recursive SQL queries has long since found its way into mainstream database implementations. An adoption of these facilities could allow for the efficient coverage of recursive functions in a loop-lifted compilation. (Support for this functionality is planned for a future release of the Pathfinder XQuery compiler. Until then, the compiler "escapes" to a 1:1 mapping of user-defined

[10] See `http://msdn.microsoft.com/netframework/future/linq/` for a recent version.

functions into procedure declarations in Pathfinder's back-end interface language MIL [Boncz99].)

Getting the Most out of Loop-Lifting

The relational algebra dialect emitted by our compiler is sufficiently simple to be efficiently implemented on any relational back-end. The way the algebra operators are combined in loop-lifted XQuery evaluation plans, however, is quite different from the usual $\pi - \sigma - \bowtie$ query pattern found in classical database applications. As such, generated query plans often will not receive optimal support on commodity systems.

The execution plans do, however, show a number of characteristics that provide promising hooks for XQuery optimizations. While such optimizations have traditionally been performed using specialized XQuery algebras, a back-end's relational query optimizer that is made aware of such characteristics is now enabled to efficiently treat XQuery expressions in a purely relational fashion. In the upcoming chapter, we will discuss our approach to the detection of *joins* and the optimization with respect to *order*. Furthermore, we will see how loop-lifted XQuery plans may serve as a promising platform to derive *result size estimates* for arbitrary XQuery expressions.

5

The Pathfinder XQuery Compiler

In the long standing research on database query optimization, it turned out that the key to effective optimizations is the availability of a powerful algebraic rewrite framework. On the one hand, loop-lifting provides the basics for such a framework in the form of a simple relational algebra and thus makes well-established optimization techniques from the relational domain readily accessible to the processing of XQuery. On the other hand, the relational plans generated by our compiler can reach considerable size. The use of joins to implement XQuery's iteration primitive `for` results in loop-lifted plans with characteristics that are quite uncommon for relational systems. Finally, the prevalent notion of *order* in the XQuery source language makes its consideration all the more interesting for relational query optimization. Under these premises, we can hardly expect existing optimizers to implement loop-lifted query plans in the most efficient manner. Novel optimization techniques are asked for, which is what we will look into now.

As part of the MonetDB/XQuery system, the *Pathfinder* XQuery compiler provides all the tools to handle loop-lifted evaluation plans efficiently. Though initially driven by the execution of XQuery, our optimizations are universal by nature and may prove equally beneficial in other application domains. We will present these optimizations in the following order. The *peephole-style* plan analysis described in Section 5.1 addresses the unusual *shape* and *size* of the loop-lifted plans and paves the ground for the consideration of *order* in Section 5.2. Section 5.3 adds an inference procedure that provides accurate *cardinality forecasts* for XQuery.

Together with our contributions from the previous chapters, these optimizations add up to the purely relational XQuery processor *MonetDB/XQuery*. Per-

formance experiments in Section 5.4 report on the system's outstanding query performance, which backs the overall viability of our approach. We look into other people's work in Section 5.5.

5.1 Logical Optimizations in Pathfinder

The compilation strategy we pursue assumes only a very restricted—and, hence, easily implementable—dialect of a relational algebra. Yet, it fully accounts for arbitrary expression nesting in XQuery. The price we pay, however, is the considerable size of the resulting algebraic code. The twenty queries from the XMark benchmark, *e.g.*, amount to several thousand operators in their relational plan trees—way beyond of what traditional optimizers are efficiently able to cope with.

5.1.1 DAGs for Loop-Lifted Query Plans

Fortunately, the emitted plan trees display a significant amount of *sharing opportunities*, particularly if the nesting depth of the original query is high. The Pathfinder compiler exploits this fact and uses *directed acyclic graphs (DAGs)* as a representation for relational plans instead. Though the size of these DAGs is still quite remarkable (53 to 500 operators for XMark queries), this rewrite takes Pathfinder's execution plans into a manageable domain.

Besides the obvious reduction of algebraic plan sizes, we also found plan DAGs to elegantly reflect the correct handling of *side-effects* in the XQuery language (cf. Section 4.4.3). Pathfinder lists each occurrence of an XQuery node constructor exactly once in the DAG. This way, we instantiate precisely as many new node identities as demanded by the XQuery semantics.

The most striking benefit, however, is the possibility to easily track the *data flow* in the algebraic DAG. Pathfinder performs data flow analysis in a novel, peephole-style [McKeeman65] fashion, as described next.

5.1.2 A Peephole-Style Plan Analysis

The use of *pattern matching* on algebraic plan trees has proven quite a successful approach to query optimization in classical database systems. The holistic view that is required for the involved matching process, however, impedes its practical application for the query sizes observed in relational XQuery evaluation plans. This is why the Pathfinder compiler takes quite a different approach to the optimization of loop-lifted query plans. A *peephole-style* inspection restricted to *single* operator nodes makes the approach scale easily to the DAG sizes we consider.

Property	Description
icols: $\{a_1, \ldots, a_n\}$	columns $a_1, \ldots, a_n$ are required to evaluate some upstream operator
keys: $\{a_1, \ldots, a_n\}$	columns $a_1, \ldots, a_n$ are key candidates
$a.dom$: α	column a takes values from the domain α
$a.const$: v	column a assumes the constant value v

Table 5.1: Properties of DAG operators and their respective subtrees.

To compensate for the now restricted view on the plan, a *property inference* phase precedes the actual rewriting and carries additional information about the vicinity of each plan node *into* the operator itself. The information we need for rewriting is captured in a set of annotations that we infer for each operator during a single DAG walk. The most important operator properties inspected by our compiler are listed in Table 5.1. We will elaborate on their semantics and use in the following. For a detailed discussion of peephole-style optimization, we refer to [Grust05, Grust06].

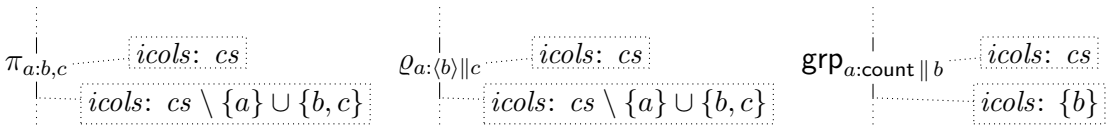
Column Dependency Analysis with *icols*

To minimize the size of intermediate results in relational plans, it is usually desirable to drop irrelevant columns as early as possible from the execution pipeline, a technique that is often referred to as *projection pushdown* [Jarke84]. In our property-driven optimization framework, the *icols* information assigned to each plan operator serves to achieve the same effect.

Inference of *icols*. For each operator $\otimes$, we record the set of *strictly required input columns* (short: *icols*) within its property annotation, such that $\otimes.icols = \{a_1, a_2, \ldots\}$ contains all columns a_i that are strictly required to evaluate some upstream operator of $\otimes$. If, in turn, $\otimes$ itself produces the values of an attribute a_j, this column will not be listed in the *icols* set of $\otimes$'s children in the DAG. The *icols* information are propagated in a top-down traversal of the DAG. During the traversal, some algebra operators add or remove columns from the propagation set. To illustrate this process, we sketched the *icols* inference for some frequently occurring plan operators in Figure 5.1.

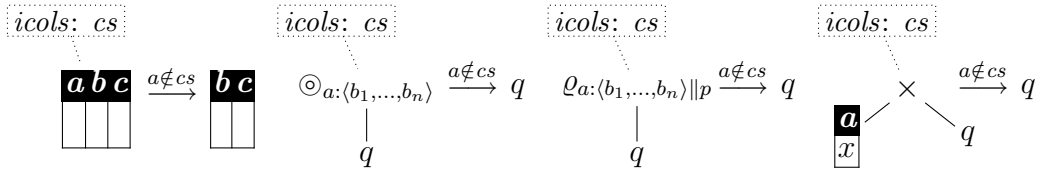
We seed the inference of *icols* at the DAG's root with $icols = \{pos, item\}$, as those are the columns needed to serialize the overall query result back into XML.

Exploiting the Information on Strictly Required Columns. The early introduction of *projection* operators into the plan DAG is the most apparent use



(a) Projection operator π. (b) Row numbering. (c) Aggregation.

Figure 5.1: Top-down inference of *icols* sets for the three selected algebra operators. DAG annotations denoted by ⋯ (*cs*: set of column names).



(a) Literal table. (b) Arithmetics. (c) Row numbering. (d) Cart. product.

Figure 5.2: Peephole-style rewrites based on the *icols* property that *eliminate* operators from the DAG. Literal table in (d) contains exactly one tuple.

of the information on *icols*. With the help of the annotation, the rewrite decision to do projections as early as possible now depends on the inspection of only a single plan node. The same information may also be used, however, to *eliminate* an algebraic operator altogether if its output is never inspected by any upstream operator. There are several spots in our algebra where such optimizations might apply. The peephole-style rewrite rules in Figure 5.2 illustrate a number of cases where this idea becomes effective.

The output of our XQuery compilation turns out to be highly susceptible to optimizations of this kind. For example, to evaluate a sequence of XPath location steps, the generated plans establish a new *pos* column in each intermediate step result. While this ensures the compositionality of our compilation procedure, the new *pos* column will never be required by any subsequent location step. We already found this opportunity appealing for *XPath step bundling* in Chapter 4. *icols*-based rewrites using the rule in Figure 5.2(c) provide an efficient implementation for XPath step bundling and may indeed lead to a significant plan reduction. In case of the XPath-centric XMark query *Q15*, *e.g.*, the exploitation of *icols* cut down the plan size from 100 to only 32 operators in Pathfinder.

We will unleash the full power of *icols*-based rewrites in Section 5.2, where we look into plan optimizations that relate to *order*.

Information on Key Columns and Value Domains

It is a common situation in loop-lifted XQuery evaluation plans that columns are populated with (artificial) key values. This happens most notably in cases where the row numbering operator ϱ (or its unsorted counterpart #) establishes a new column with unique row numbers. The *keyness* of such columns is easy to infer in a bottom-up DAG walk and we record the respective information in the operator's *keys* property.

The propagation of the *keys* property is described by a number of inference rules. Key joins, *e.g.*, will always propagate keyness bottom-up:

$$\frac{a \in e_1.keys \qquad b \in e_2.keys}{(e_1 \bowtie_{a=b} e_2)\,.keys\colon e_1.keys \cup e_2.keys}\ . \tag{5.1}$$

The establishment of a new column numbering also marks the spot where our compiler records the *active domain* of the values taken by the new column. Obviously, the actual *value* of an expression is not known before query evaluation time. We may, however, introduce *abstract domain identifiers* $\alpha, \beta, \ldots$ instead and assign them to a column a of an operator $\otimes$ in the DAG (*e.g.*, $\otimes.a.icols\colon \alpha$). Furthermore, our compiler infers domain *inclusion* information. From an inclusion $\alpha \subseteq \beta$, we can then conclude the inclusion of the corresponding active domains.

Much like the *keys* information, *dom* is inferred bottom-up in the plan DAG. To exemplify the procedure, the inference rule

$$\frac{e_1.a.dom\colon \alpha \qquad e_2.b.dom\colon \beta \qquad \beta \subseteq \alpha}{(e_1 \bowtie_{a=b} e_2)\,.a.dom = \beta \qquad (e_1 \bowtie_{a=b} e_2)\,.b.dom = \beta} \tag{5.2}$$

implemented in our compiler comprises the knowledge that each tuple in e_2 will always find a matching partner in e_1 for the join $\bowtie_{a=b}$, since the active domain α of $e_1.a$ includes β, the domain of $e_2.b$. For a larger set of *keys* and *dom* derivation rules, we refer the reader to [Grust05].

To illustrate the property derivation process, Figure 5.3 displays the DAG annotations *icols* and *keys* for the (simplified) plan DAG[1] of the query

```
for $a in doc("auction.xml")/descendant::open_auction
  where $a/initial lt 180                                    (Q1)
  return $a .
```

Observe how we could, *e.g.*, remove the row numbering operator $\varrho_{pos:\langle item\rangle \| iter}$ marked with (!) from the plan DAG, because the newly generated *pos* column is

[1] Algebra operators `DOC` and `VAL` implement the XQuery functions `fn:doc()` (access to the document relation `doc`) and `fn:data()` (typed value extraction from XML nodes), respectively.

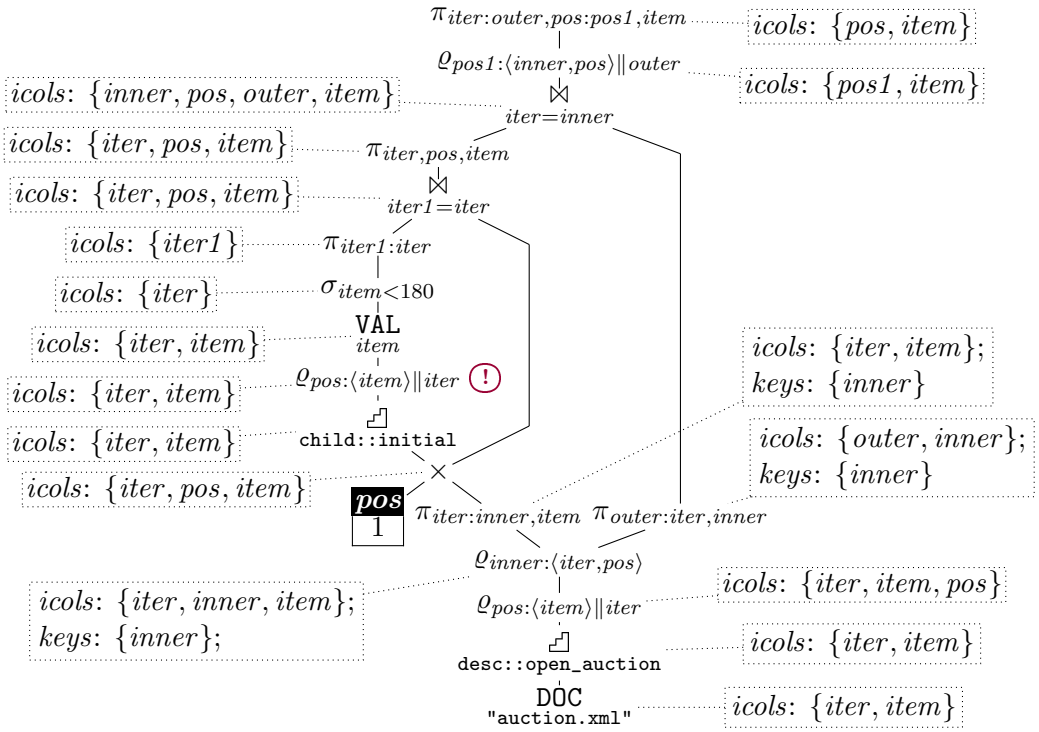


Figure 5.3: DAG annotations *icols* and *keys* for Query Q_1.

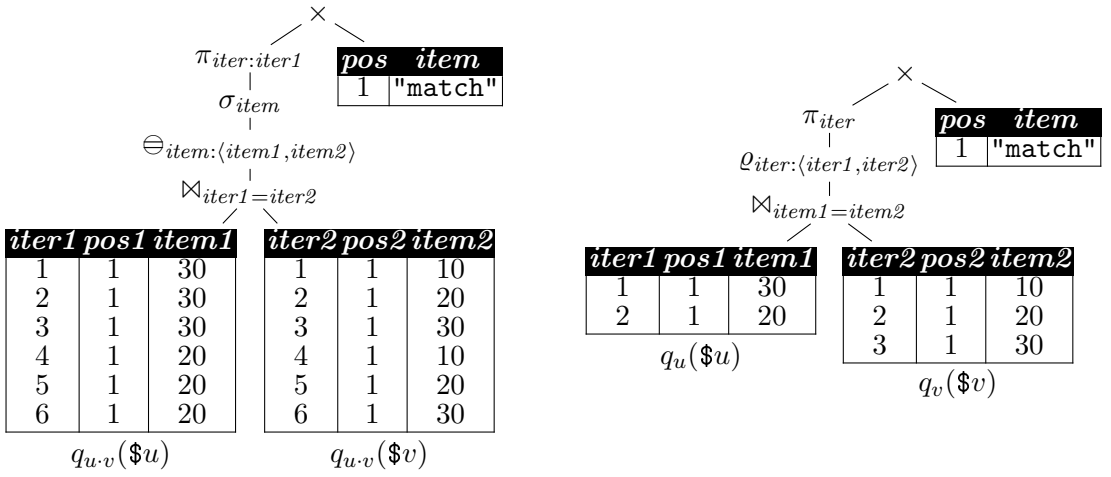
not among the operator's *icols*. For space reasons, the plan in Figure 5.3 does not include domain information (property *dom*). A second instance of the same plan in Section 5.3 will provide this information.

Columns that Evaluate to a Constant Value

As the last DAG annotation mentioned in this section, property *const* identifies columns that are guaranteed to assume a constant value v. Such columns may be the effect of literal values in the original query (compiled via Rule Const) or originate from the compilation procedure itself (*e.g.*, the constant *pos* columns established by Rule For). The inference of the column property *const* will *propagate* such constants to their actual use in, *e.g.*, value comparisons with $\circledcirc$ and may avoid the need to materialize the column in the final execution plan.

5.1.3 Robust XQuery Join Detection

A further-reaching optimization that may be achieved in terms of peephole-style plan rewriting has been described by Grust [Grust05]. This work particularly looks into *join situations* in XQuery. For lack of an explicit join syntax in the XQuery



(a) Loop-lifted plan for Q_2.

(b) Equivalent join plan.

Figure 5.4: Relational evaluation of XQuery example Q_2. (a) Loop-lifting effectively establishes a Cartesian product of both operands. (b) The *independent* representation of $\$u$ and $\$v$ in the join plan avoids this product instead.

language, join semantics is typically expressed in terms of nested `FLWOR` clauses or equivalent path expressions. The `where` clause in the query

$$s\left\{\begin{array}{l}\texttt{for \$}u\texttt{ in (30, 20) return}\\ s_u\left\{\begin{array}{l}\texttt{for \$}v\texttt{ in (1, 2, 3) return}\\ s_{u\cdot v}\left\{\begin{array}{l}\texttt{where \$}u\texttt{ eq \$}v\texttt{ * 10}\\ \texttt{return "match" ,}\end{array}\right.\end{array}\right.\end{array}\right. \qquad (Q_2)$$

for example, makes the nested `for` clauses encode a logical join over the input sequences `(30, 20)` and `(1, 2, 3)`.

A straightforward implementation of this query as a nested iteration, however, seems far from efficient. Quite the contrary, the loop-lifted evaluation of Query Q_2 will effectively compute the *Cartesian product* of the two input sequences as illustrated in the plan excerpt in Figure 5.4(a). With a relational database system in the back, we obviously could do better. The *independent* evaluation of both join operands followed by the *value-based* join $\bowtie_{item1=item2}$ avoids the computation of the Cartesian product and makes the join accessible to the RDBMS instead (Figure 5.4(b)). Ideally, the relational kernel will back this plan with efficient hash or merge join implementations.

The effectiveness of the plan rewrite in Figure 5.4 is high. In [Boncz06b], we evaluated both plan variants for the join queries in the XMark benchmark

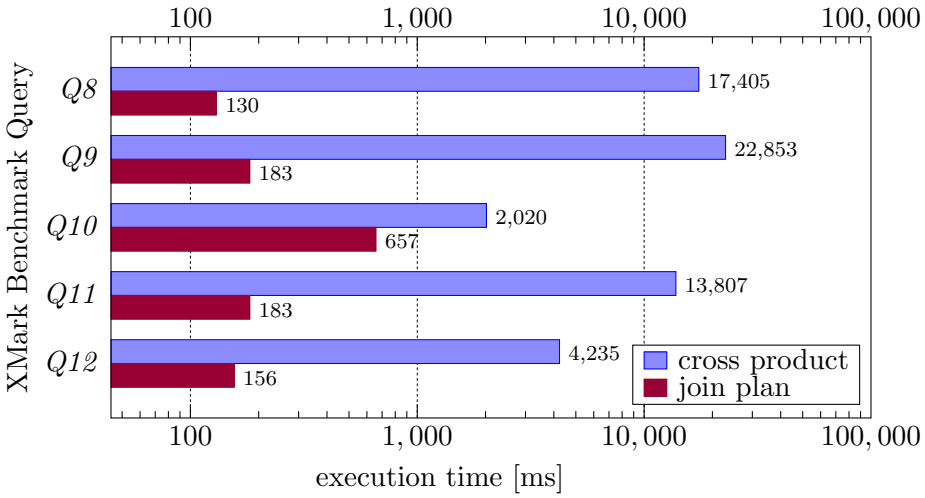


Figure 5.5: XQuery join optimization. Rewriting loop-lifted XQuery evaluation plans into equivalent join plans significantly improves query performance, even on a moderately sized XMark instance (11 MB). Experiments conducted in [Boncz06b].

set (Queries *Q8–Q12*) using the 0.10.2 version of MonetDB/XQuery. Figure 5.5 documents a performance improvement of up to two orders of magnitude even on an XML instance of moderate size (11 MB). Given the quadratic complexity of a nested loops evaluation, this indicates that the rewriting of join queries is an absolute *must* for the support of large-sized XML documents.

However, a dependable *detection* of join scenarios in a given XQuery expression constitutes quite a challenge to a system's query optimizer. An analysis of the input expression itself is only a half-hearted solution for the problem and turns out to be rather fragile with respect to syntactical variations of the query. In fact, none of the XQuery systems we investigated in [Boncz06b] was able to reliably recognize all of the five join queries in XMark.

In contrast to that, the Pathfinder optimizer relies on the join detection technique proposed by Grust [Grust05]. It constitutes a peephole-style rewrite strategy which detects join opportunities in loop-lifted XQuery plans in a purely algebraic fashion. The key to the detection is the inference of an *independence* property in the algebraic plan DAGs. To this end, the optimizer derives the degenerate *multivalued dependency* $\varnothing \twoheadrightarrow a_1, \ldots, a_n$ for a plan operator $\otimes$ whenever columns $a_1, \ldots, a_n$ are independent of all remaining columns in the output of $\otimes$. It is exactly this independence that allowed us to compute the representations $q_u(\$u)$ and $q_v(\$v)$ for $\$u$ and $\$v$ in Query Q_2 in separation (cf. Figure 5.4(b)).

Once our compiler has recognized an independence of this kind, it makes the situation explicit in the plan and introduces a Cartesian product $\times$. The resulting

combination $\sigma - \ominus - \times$ is then easily realized as a relational join in a subsequent step and rewritten accordingly. For further details, refer to [Grust05].

5.2 The Importance of Order

We have seen in the foregoing chapters that *order* is an integral part of the XQuery semantics. The XPath accelerator encoding in Chapter 2, *e.g.*, took special care to implement *document order* in terms of the preorder rank $pre(v)$, as did the staircase join operator in Chapter 3 when it came to retrieving the result of an XPath location step. Likewise, in Chapter 4, columns *iter* and *pos* of a loop-lifted sequence representation implemented *iteration* and *sequence order* by relational means. Indeed, all three notions of order are quite pervasive in the compilation strategy we devise.

5.2.1 Order in Loop-Lifted XQuery

The pervasive presence of order constraints in our algebra code is reflected in the frequent appearance of the *row numbering operator* ϱ in the relational plan DAG, a consequence of the inherently *unordered* data model of relational systems. In the query execution plan, each such ϱ operator will impose a specific physical ordering on its input, such that loop-lifted XQuery plans will either *(i)* involve a large number of explicit sort operators or *(ii)* enforce the propagation of a specific ordering throughout the physical execution. Both situations obviously disrupt the unordered evaluation model of the relational system and may seriously impair performance.

Row Numbering: An Indicator for Order Constraints

On the plus side, the row numbering operator ϱ makes order effects easily recognizable in the *logical* plan DAGs inspected by our query optimizer. Hence, we will use the occurrence of ϱ as a reasonable *indicator* for order constraints in loop-lifted XQuery plans and strive for its avoidance wherever possible.

We have already encountered one instance where a peephole-style optimization was able to reduce the occurrences of order constraints: the exploitation of *icols* information led to the removal of interleaving ϱ operators in successive XPath location steps (which we referred to as "XPath step bundling"). We have mentioned the significant cut-down of the overall plan size for the XPath-centric XMark query *Q15* as a consequence of *icols* analysis on page 122. At closer look, the size reduction mainly results from the elimination of expensive ϱ operators from the plan: out of 27 ϱ operators in *Q15*, *icols* examination removed 25.

5.2.2 Order Indifference in XQuery

The avoidance of row numbers in intermediary path expression results corresponds directly to the semantics of XPath: the sequence order within a context set is *immaterial* to the outcome of an XPath location step. There are, however, far more situations in XQuery where order does not affect the evaluation of an expression:

(i) the quantifiers `some` and `every`,

(ii) the existential semantics of general comparisons (`=`, `<`, ...),

(iii) aggregate functions (`fn:max()`, `fn:count()`, ...),

(iv) further built-in functions (`fn:empty()`, `fn:exists()`, ...),

(v) `FLWOR` expressions whose result is explicitly re-ordered by an `order by` clause, and, most notably,

(vi) the explicit relaxation of order constraints in terms of XQuery's *ordering mode* (keywords `ordered{·}` and `unordered{·}`) and the built-in function `fn:unordered()`.

Order semantics is deeply wired into the XQuery language. Therefore, the *indifference* of order in the above situations turns out to be hard to grasp in XQuery itself [Grust06]. In fact, the W3C XQuery specifications explicitly omit a formal description of order in their Formal Semantics [Draper05, § 4.8].

Algebraic Order Indifference

It turns out, though, that in all of the above cases, the indifference of order is directly observable in loop-lifted algebraic code:

(i) if an expression is indifferent with respect to *sequence order*, its corresponding code may populate column *pos* with *arbitrary* (though unique) values. Likewise,

(ii) if an expression does not depend on *iteration order*, column *iter* may be filled with an *arbitrary* unique numbering.

In these situations, we can make efficient use of the *unsorted* row numbering operator # in our algebra. Using this operator, we can instruct the database system to behave precisely as we need it: $\#_a$ introduces a new column a, filled with arbitrary unique values. The operation is easy to perform: a simple numbering according to the existing physical tuple order serves the purpose equally well as the re-use of existing tuple identifiers (*row ids*) as the new column (cf. Section 4.1.2).

Note that we may *not* simply drop the respective columns during the generation of the plan. Since the "compiles to" function $\cdot \Rightarrow \cdot$ describes a fully compositional procedure where expressions are allowed to nest arbitrarily, any compiled subexpression is assumed to consume and produce relations with columns *iter* and *pos* in place as usual.

Compilation in Awareness of Order Indifference

The trade of costly row-numbering operators ϱ for inexpensive # operators only requires a minor modification to the rule set of Pathfinder. The prototypical example is the compilation rule for `fn:unordered ()` that literally implements the disposal of sequence order and the population of column *pos* with arbitrary values instead:

$$\frac{\Gamma; \mathsf{loop} \vdash e \Rightarrow (q_e, \Delta_e)}{\Gamma; \mathsf{loop} \vdash \texttt{fn:unordered}\,(e) \Rightarrow (\#_{pos}\,(\pi_{iter,item}(q_e))\,, \Delta_e)}\ . \quad \text{(Fn:Unordered)}$$

As this rule "overwrites" the position information of q_e, column *pos* is no longer a required input column of q_e (in the sense of Section 5.1.2). In effect, the *icols* column dependency analysis will avoid the construction of *pos* in expression q_e outright, which is likely to trigger the removal of an instance of ϱ downstream in the algebraic plan.

Compilation rules for quantifiers, aggregates, and built-in functions disregard position information in a similar fashion. In Section 4.5.2, we dropped sequence order before aggregating type information in order to implement XQuery sequence type matching (page 105). As mentioned before, the algebra code for XPath location steps (see Rule Step) removes column *pos* from the context set before evaluating the step operator $\leftthreetimes$. All these cases will naturally lead to a disposal of ϱ in the algebraic plan DAGs downstream.

Exploiting XQuery's *ordering mode*

In awareness of the performance improvements to gain,[2] the W3C XQuery Candidate Recommendation provides an explicit means to *relax* the ordering constraints of XQuery even further. In dependence of the *ordering mode* setting (controlled in terms of the `ordered { · }` and `unordered { · }` clauses), an XQuery implementation is free to

(i) return node sequences in any order as the result of XPath navigation expressions and

[2] Quoting from [Boag05, § 3.9]: *"For expressions where the ordering of the result is not significant, a performance advantage may be realized by setting the ordering mode to* `unordered`*, thereby granting the system flexibility to return the result in the order that it finds most efficient."*

$$\frac{\begin{array}{c} \textit{ordering mode} = \texttt{unordered} \\ \{\ldots, \$v_i \mapsto (q_{v_i}, \Delta_{v_i}), \ldots\}; \mathsf{loop} \vdash e_1 \Rightarrow (q_1, \Delta_1) \qquad q_i \equiv \#_{inner}(q_1) \\ \mathsf{loop} \equiv \pi_{iter:inner}(q_i) \qquad \mathsf{map} \equiv \pi_{outer:iter,inner}(q_i) \qquad q_v \equiv \boxed{\begin{array}{c}\textbf{\textit{pos}}\\1\end{array}} \times \pi_{iter:inner,item}(q_i) \\ q_p \equiv \boxed{\begin{array}{c}\textbf{\textit{pos}}\\1\end{array}} \times \pi_{iter:inner,item}\left(\varrho_{item:\langle pos\rangle \| iter} \pi_{inner,iter,pos}(q_i)\right) \\ \Gamma_v \equiv \{\ldots, \$v_i \mapsto (\pi_{iter:inner,pos,item}\left(q_{v_i} \bowtie_{iter=outer} \mathsf{map}\right), \Delta_{v_i}), \ldots\} \\ + \{\$v \mapsto (q_v, \Delta_1)\} + \{\$p \mapsto (q_p, \emptyset)\} \\ \Gamma_v; \mathsf{loop}_v \vdash e_2 \Rightarrow (q_2, \Delta_2) \end{array}}{\begin{array}{c} \{\ldots, \$v_i \mapsto (q_{v_i}, \Delta_{v_i}), \ldots\}; \mathsf{loop} \vdash \texttt{for}\ \$v\ \texttt{at}\ \$p\ \texttt{in}\ e_1\ \texttt{return}\ e_2 \Rightarrow \\ \left(\pi_{iter:outer,pos:pos_1,item}\left(\varrho_{pos_1:\langle inner,pos\rangle \| outer}\left(q_2 \bowtie_{iter=inner} \mathsf{map}\right)\right), \Delta_2\right) \end{array}} \quad (\textsc{For}^{\#})$$

Figure 5.6: Compilation of `for`-loops if *ordering mode* = `unordered`.

(ii) iterate over a `FLWOR` body without adhering to the sequence order of the binding sequence

if *ordering mode* is set to `unordered`. Our compiler acknowledges this opportunity to let go of order by extending the respective compilation rules by a premise specifying the setting of *ordering mode*. For instance, in Rule $\textsc{Step}^{\#}$

$$\frac{\textit{ordering mode} = \texttt{unordered} \qquad \Gamma; \mathsf{loop} \vdash e \Rightarrow (q_e, \Delta_e)}{\Gamma; \mathsf{loop} \vdash e/\alpha::\nu \Rightarrow \left(\#_{pos}\left(\delta\left(\left(\pi_{iter,item}(q_e)\right) \between_{\alpha,\nu} \Delta_e\right)\right), \Delta_e\right)}\ , \qquad (\textsc{Step}^{\#})$$

our compiler recognizes the indifference of order for the result of an XPath location step in the `unordered` *ordering mode*. The use of # to establish the *pos* column of the result makes this indifference explicit to the relational back-end.

Likewise, the sequence order of the binding sequence of an XQuery `for` construct no longer determines iteration order if *ordering mode* = `unordered`. The interaction between these two notions of order led to the application of ϱ in Equation 4.2 (page 84) where we derived the relational encoding $q_{x\cdot y}(\$v_{x\cdot y})$ of a binding variable $\$v_{x\cdot y}$. If *ordering mode* is set to `unordered`, we may instead populate column *inner* with arbitrary values and represent $\$v_{x\cdot y}$ as

$$q_{x\cdot y}(\$v_{x\cdot y}) \equiv \boxed{\begin{array}{c}\textbf{\textit{pos}}\\1\end{array}} \overset{\times}{\diagup\ \diagdown} \begin{array}{c}\pi_{iter:inner,item}\\ \#_{inner}\\ q_x(e_{x\cdot y})\end{array}\ . \qquad (5.3)$$

The same idea is incorporated in compilation rule $\textsc{For}^{\#}$ that accounts for order-indifferent `for` constructs in the implementation of Pathfinder (Figure 5.6).

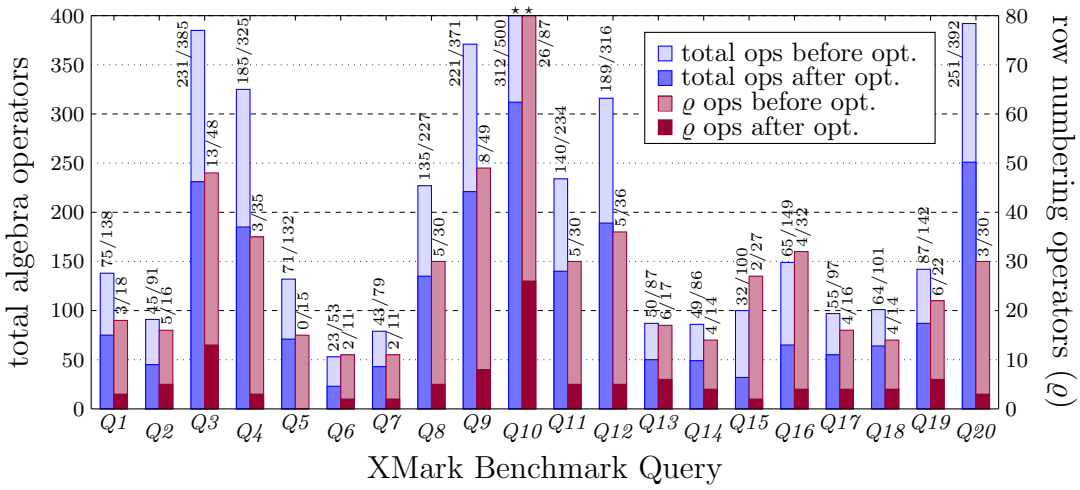


Figure 5.7: Effect of order-aware plan rewrites. Total number of DAG operators before/after optimization and respective number of row numbering operators (note the different scales on the y-axis).

5.2.3 A Performance Advantage *can* be Realized

The directly observable effect of the exploitation of such order indifferences is the avoidance of row numbering operators in the logical plan DAGs. The impact is quite significant: rewriting cuts down the number of ϱ operators in the 20 XMark queries by more than 80 % on average. To illustrate, we compiled all 20 queries with and without order-related optimizations in effect and recorded the total number of operators as well as the number of row numbering operators ϱ only. Figure 5.7 compares these numbers before and after order-sensitive plan rewrites. The logical optimizer clearly serves the purpose it was designed for.

An impact on query execution times, however, will hardly be visible, unless *physical* evaluation plans take explicit advantage of the relaxation of order constraints, as discussed next.

5.2.4 Physical Optimization and Order Awareness

While the occurrence of ϱ operators gives us a reasonable indicator for order constraints on the logical algebra level, the actual tuple ordering is ultimately determined when Pathfinder converts the algebra DAG into a physical execution plan. To keep the required sort operations at a minimum, the compiler carefully inspects physical order properties during the generation of the physical plan. This way, our planner *realizes* the performance gain that we prepared before when we substituted ϱ for #.

Our approach is quite similar to the one taken in System R [Selinger79]. Pathfinder's execution plan generator enumerates available *candidate plans* in a bottom-up fashion, each of them annotated with an expected *execution cost* as well as with information on the *physical orderings* that the plan guarantees. Of all plans, only the cheapest possible plan is retained for each physical ordering. The planner will instantly exclude the others from the generation process following the principles of dynamic programming.

Given the importance of order for the evaluation of XQuery, we exercise specific care to make this compilation phase effective. To this end, Pathfinder considers a number of physical operators in the MonetDB back-end that provide extended guarantees with respect to order. Most notably, we incorporated new implementations for the disjoint union $\dot\cup$, row numbering ϱ, and sorting operators into the system.

Disjoint Union. In addition to the usual `AppendUnion` which simply pastes its input relations one after another, MonetDB's `MergeUnion` accounts for the typical use of the $\dot\cup$ operation in loop-lifted XQuery plans. Parameterized with a physical ordering O that must be provided by both input subplans e_1 and e_2, `MergeUnion` processes both tuple streams in a parallel fashion. The output contains all tuples from e_1 and e_2 and retains the order defined by O. Furthermore, tuples from e_1 will precede entries from e_2 in the result if both are indistinguishable with respect to O.

Among other useful applications, `MergeUnion` seamlessly integrates into our compilation of XQuery's sequence construction operator `(·,·)`. As mentioned in Section 4.1.5, the use of `MergeUnion` for the disjoint union in Rule SEQ avoids the expensive sort required to implement the subsequent row numbering with ϱ (assuming properly sorted input relations).

Row Numbering. An obvious approach to implement the row numbering operator $\varrho_{a:\langle b_1,\ldots,b_n\rangle \,\|\, p}(q)$ is to physically sort the entire input relation q according to the ordering $\langle p, b_1, \ldots, b_n\rangle$ (with the partitioning attribute p as the primary sort key). The `MergeRowNumber` operator then fills column a with a counter that starts at 1, is incremented at each tuple and reset to 1 whenever a new p value is encountered.

MonetDB's alternative, `HashRowNumber`, creates a hash table on column p and initializes a counter in each bucket with the value 1. Then, for each tuple in q, the counter is looked up, emitted into the result, and incremented by 1. This implementation requires a less restrictive ordering on the input relation q, where we only demand all tuples with the same value in the partitioning attribute p to be ordered according to $\langle b_1, \ldots, b_n\rangle$. This generalization of order conditions (denoted $\langle b_1, \ldots, b_n \,\|\, p\rangle$) turns out to have other valuable applications in the Mon-

etDB/XQuery system as well, for which we refer to [Boncz05a].

Sorting. Instead of applying a full `StandardSort` to enforce a certain ordering of a relation q, there are many situations in loop-lifted evaluation plans where `RefineSort` provides a sufficient alternative. This MonetDB operator exploits the knowledge that the input relation is already sorted on a major subset of the required order and implements the refinement without fully blocking the processing pipeline.

5.3 Cardinality Forecasts for Loop-Lifted Plans

We have assumed the presence of a sound cost model for algebraic query plans in the previous section. Such cost models have been studied extensively for relational query languages and plug in quite seamlessly into the plan generator of Pathfinder.

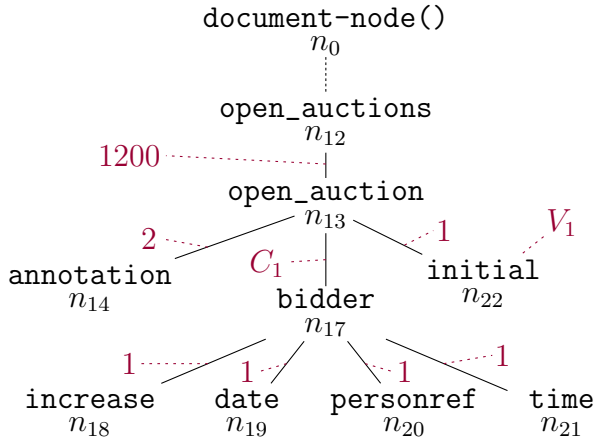
The crux of these cost models, however, is their dependence on accurate *cardinality forecasts* for the involved query expressions. The derivation of such information is well understood in the relational domain. The domain of XQuery, however, still lacks a convincing means to estimate the cardinality of arbitrary XQuery (sub-)expressions. Existing work in the domain, if any, remains limited to a subset of XPath.

The compilation procedure we devise translates XQuery expressions of arbitrary shape into purely relational query plans. As such, the adaption of estimation techniques from that domain seems a promising leverage point to achieve meaningful cardinality forecasts for relational XQuery evaluation plans. The approach we discuss here is twofold:

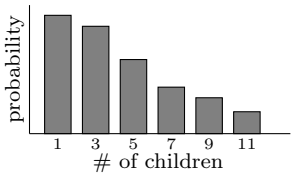
(i) Statistical guide.
In order to maintain statistical information on XML documents, we introduce *statistical guides*. A statistical guide is a strong DataGuide in the sense of [Goldman97], annotated with statistics on the underlying data.

(ii) Cardinality forecasts.
In Section 5.3.2, we will extend the property framework of our peephole optimizer to embrace the derivation of cardinality forecasts for algebraic plans in a seamless fashion.

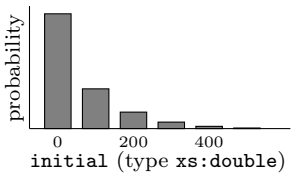
Both techniques are currently under development in the MonetDB/XQuery system and are expected to significantly improve the accuracy of cost estimations in the XQuery compiler Pathfinder.



(a) Statistical guide.



(b) Children histogram C_1.



(c) Typed value histogram V_1.

Figure 5.8: Statistical guide for a 12 MB XMark instance (excerpt). The edge $n_{13} \rightarrow n_{17}$ is annotated with the children histogram C_1, node n_{22} with the value histogram V_1.

5.3.1 Statistical Guide

The difficulty of predicting cardinalities for XQuery expressions is caused by the possibility to simultaneously query a document's shape *and* content in XQuery. We acknowledge this fact by means of a *statistical guide* that summarizes structural information in terms of child element distributions and also records the distribution of *typed values* associated with the nodes in an XML tree (as defined by the W3C XQuery Data Model [Fernández05]). A statistical guide (see Figure 5.8 for an example) implements a *strong DataGuide* in the sense of [Goldman97]. As such it is easily constructed in a single tree traversal, *e.g.*, during XML document shredding.

We capture structural information by *edge* annotations in the statistical guide. A guide edge corresponds to multiple edges in the input document. By default, we annotate guide edges with the average number of edges sharing a common parent node in the document. To exemplify, in the document summarized in Figure 5.8, an `open_auction` element (n_{13}) contains two child elements tagged `annotation` (n_{14}) on average. For selected guide edges, such annotations may be refined in terms of a *children* histogram that reflects the actual children distribution more accurately. The edge $n_{13} \rightarrow n_{17}$, for example, has been refined by the children histogram C_1 (Figure 5.8(b)).

Selectivity estimations for value-based predicates are supported in terms of *typed value histograms* that the database administrator may decide to set up for specific nodes in the statistical guide. The histogram V_1 in Figure 5.8(c) sketches the value distribution for node n_{22} (tag `initial`) in our example guide (assuming the corresponding nodes in the document are of type `xs:double`).

Observe that the statistical guide itself takes the shape of a tree. As such, it smoothly integrates with our storage of XML documents and could, *e.g.*, be maintained in terms of a *pre/post* encoding. As Goldman *et al.* [Goldman97] point out, the size of strong DataGuides usually remains small even for highly irregular XML trees. For data-centric XQuery applications, we can even expect that the statistical guide easily fits into main memory, where it is efficiently accessible to the query compiler.

5.3.2 Cardinality Forecasts

In line with the property-driven optimization process we discussed in Section 5.1, we introduce cardinality estimates in terms of the additional property *card*, which, annotated to an operator $\otimes$, represents a forecast of the cardinality of the output relation of $\otimes$.

Guide Nodes and Cardinality Inference

For the inference of *card*, we rely on the presence of the statistical guide as well as a further new concept, *guide nodes* (recorded for a column c in terms of the annotation *c.guide*). A guide node property *c.guide*: n_i indicates that column c contains surrogates of nodes that correspond to node n_i in the statistical guide (recall the n_i node identifiers in Figure 5.8).

To illustrate the inference process for both properties, Figure 5.9 shows the annotated plan DAG for Query Q_1 in Section 5.1.2:

```
for $a in doc("auction.xml")/descendant::open_auction
  where $a/initial lt 180                                  (Q1)
  return $a .
```

Guide Nodes. In the DAG in Figure 5.9, operator `DOC` ① implements the access to the persistent document storage and returns a singleton $\langle iter, pos, item\rangle$ relation in which column *item* carries the surrogate of the document node of `"auction.xml"`. For this relation, we record a reference to the root n_0 of the statistical guide in *item.guide*.

Guide node annotations are usually propagated bottom-up in the operator DAG (*e.g.*, *item.guide*: n_{13} from ② to ③). The *guide* annotation changes whenever

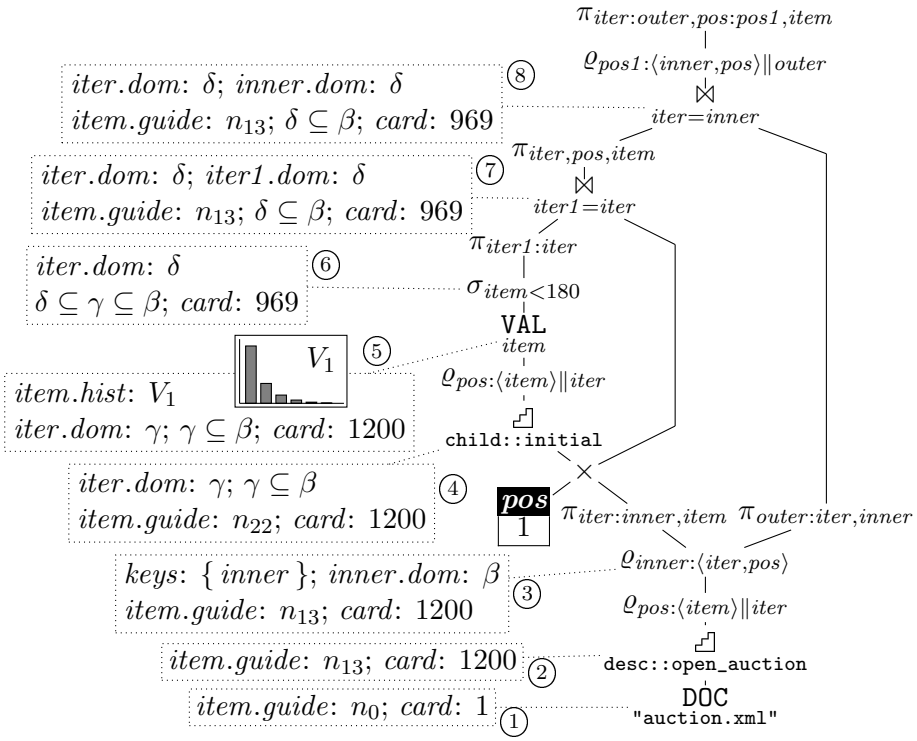


Figure 5.9: Plan DAG for query Q_1, annotated with inferred properties, guide nodes (property *guide*), and expected cardinalities (*card*) for a 12 MB XMark instance (annotations refer to Figure 5.8).

a staircase join operator is encountered during the propagation. If ⌐ performs a step along axis α, we also step along α in the guide to keep the *guide* property up to date. In Figure 5.9, the location step `descendant::open_auction` makes n_{13} the new guide node when moving from ① to ②. In this case, we may infer from the edge annotations in the statistical guide that ⌐ at ② will yield 1200 nodes (thus, *card*: 1200).

Cardinality Inference. The guide node-based cardinality forecasts are accompanied by cardinality inference rules that extend the rule set we sketched in Section 5.1.2. Typically, these rules rely on further plan properties to infer accurate cardinality estimates. Quite similar to Rule 5.2 (which we used to propagate domain information across a join), the rule

$$\frac{a \in e_1.keys \qquad e_1.a.dom\colon \alpha \qquad e_2.b.dom\colon \beta \qquad \beta \subseteq \alpha \qquad e_2.card\colon c_2}{(e_1 \bowtie_{a=b} e_2)\,.card\colon c_2} \tag{5.4}$$

incorporates the knowledge that each tuple in e_2 will find exactly one join partner in e_1 for the join $\bowtie_{a=b}$, since column a is key in e_1 and contains all values of column b in e_2 (due to $\beta \subseteq \alpha$). In the DAG of Figure 5.9, we used this rule to propagate *card* across the key joins at ⑦ and ⑧.

Typed Value Access. The `where` clause in Query Q_1 uses the value comparison operator `lt` to compare an XML tree node (the result of the subexpression `$a/initial`) to the integer `180`. In such situations, *atomization* is implicitly performed to extract the typed value of the involved node.

In the plan DAG in Figure 5.9, operator `VAL` at ⑤ marks the spot where the system trades nodes for atomic values (of type `xs:double` here). Column *item* of the output relation will thus contain the 1200 typed values that correspond to the `initial` nodes (guide node n_{22}) of the operand of $\texttt{VAL}_{item}$. At ⑤, we will remove the *guide* annotation n_{22} from the *item* column and annotate the typed value histogram V_1 associated with n_{22} to column *item* (property *hist*) instead. This process is captured by the inference rule

$$\frac{e.a.guide\colon n \qquad n.hist\colon V_n}{(\texttt{VAL}_a\, e)\,.a.hist\colon V_n}\ , \tag{5.5}$$

where $n.hist\colon V_n$ denotes that node n in the statistical guide has been refined with the typed value histogram V_n.

The value distributions provided by a column's *hist* annotation may then be used to estimate predicate selectivities in the classical relational style (*e.g.*, at ⑥) to properly maintain our *card* forecast. In the example in Figure 5.9, we ultimately estimate that 696 `open_auction` elements ($item.guide\colon n_{13}$) will be returned by Query Q_1. The actual evaluation of Q_1 yields 986 nodes—the forecast is thus not too far off.

Experiments [Sakr06] indicate that the use of loop-lifted query plans constitutes a highly flexible and accurate approach to result size estimation for XQuery (sub-)expressions. The technique is currently under implementation in the XQuery compiler Pathfinder.

5.4 MonetDB/XQuery: A Fast and Scalable XQuery Processor

The tailor-made optimizations for loop-lifted XQuery evaluation plans contributed the final pieces to build up an efficient and standards-compliant XQuery processor based on relational database technology. The construction of such a system is

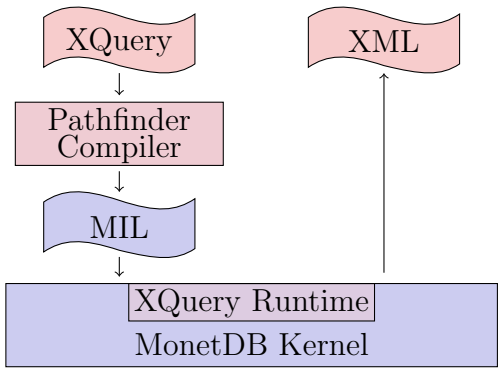


Figure 5.10: Simplified architecture of the MonetDB/XQuery system.

what we did in the context of this work in order to assess the viability of our purely relational approach.

The outcome is *MonetDB/XQuery*, one of the fastest and most scalable XQuery implementations in existence at the time of this writing. The system is backed by the MonetDB database kernel, a purely relational system tuned to exploit the capabilities of modern computing hardware [Boncz02]. Version 4.10.2 is the latest issue of the open-source kernel that comprises more than a decade of database research and development carried out at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam, The Netherlands.

XQuery support is provided by the *Pathfinder compiler* whose internals we described in this work. A small *runtime extension module* adds an implementation of staircase join to the kernel as well as a set of features that facilitate the shredding, handling, and serialization of XML documents. In addition to an implementation of the core XQuery functionalities, Pathfinder provides the *schema import*, *static typing*, *full axis*, *module* and *serialization* features described in Section 5.2 of the W3C XQuery Candidate Recommendation [Boag05]. Support for the XQuery Update Facility [Chamberlin06] is currently under implementation.

5.4.1 System Architecture

The implementation of MonetDB/XQuery follows the architecture illustrated in Figure 5.10. Pathfinder uses the loop-lifting technique to compile the user input into relational algebra and optimizes it for execution on MonetDB. The resulting plan is then emitted to the MonetDB server in terms of a MIL[3] program, where it is executed with the help of the XQuery runtime module. A serialization routine assembles the result into XML before it is shipped back to the user.

[3]MIL stands for *MonetDB Interpreter Language* [Boncz99].

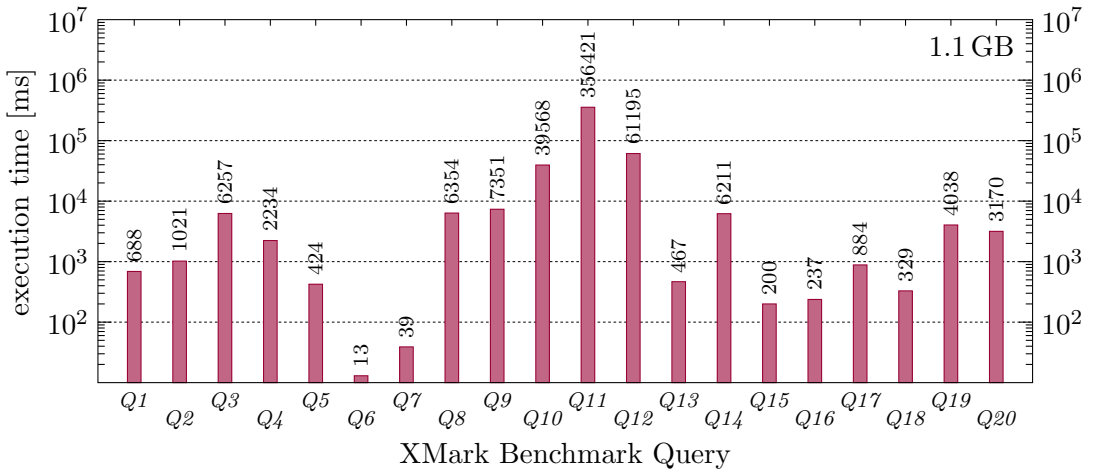


Figure 5.11: MonetDB/XQuery performance. Execution times for the 20 XMark queries based on an 1.1 GB XML instance.

The compiler is designed to be *re-targetable*: any relational system that supports the algebra dialect listed in Chapter 4 may serve as a suitable back-end to Pathfinder. Our current development work includes the emission of code for the upcoming version 5 of MonetDB [Kersten05], MonetDB/X100 [Boncz05d], and IDEFIX [Grün06].

5.4.2 Overall Query Performance

The XMark benchmark [Schmidt02] is a widely accepted means to assess the performance and capabilities of high-volume XQuery processors. We used version 0.10.2 of the MonetDB/XQuery system to run the 20 XMark queries on documents of sizes ranging from 11 MB to 11 GB. All test queries were run multiple times and execution times averaged. The system in use was a 2×3.2 GHz Intel Xeon system, equipped with 8 GB of primary storage. We ran the MonetDB kernel off a 140 GB `ext3` file system, using the version 2.6.5 Linux kernel as shipped with SuSE Enterprise Server 9. The performance results we report indicate the system's raw query execution time. They do not include query compilation time ($\approx$ 50 milli-seconds on average) and result serialization.

The system had no problems loading all documents we provided. Figure 5.11 documents the execution times we observed on a document instance of 1.1 GB size (XMark scale factor 10). MonetDB/XQuery was able to evaluate most of the 20 queries on this document size in interactive time, the only outlier being the join query *Q11* which we will look into shortly (Section 5.4.4). A large share of the queries returned after only sub-second evaluation times.

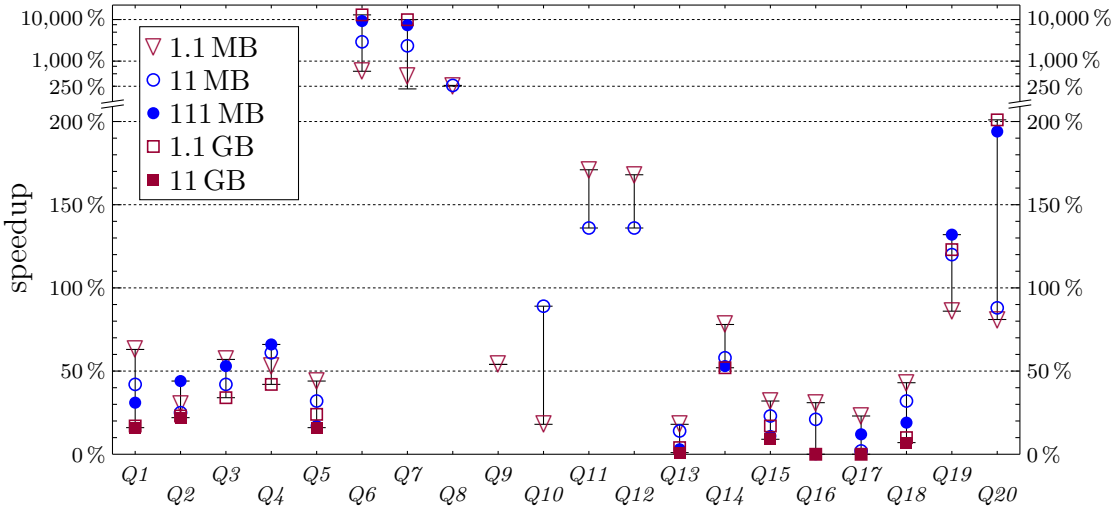


Figure 5.12: Observed impact of order indifference (speedup) on the XMark benchmark query set. The reduced sort overhead leads to performance improvements of up to four orders of magnitude.

Our experiments clearly show that the use of relational technology is a promising approach to process XQuery on high-volume data. Comparative experiments undertaken in [Boncz06b], in fact, conclude that MonetDB/XQuery is among the fastest XQuery processors currently in existence.

5.4.3 Order Awareness in Pathfinder

The Pathfinder compiler is fully aware of the order-related optimizations we discussed in Section 5.2. To assess their effect on the actual query performance of MonetDB/XQuery, we ran the 20 XMark queries with and without order-sensitive optimizations in effect. Figure 5.12 documents the speedup we observed on XML document instances ranging from 1.1 MB to 11 GB serialized size. A speedup of 100 % in this figure indicates that Pathfinder was able to generate algebraic code that executed twice as fast due to the exploitation of order indifference.

For the majority of the queries, the observed speedup falls into the range of 0 to 200 %. This demonstrates quite clearly how much processing time the system spends on unnecessary sort operations in the original plans. In [Grust06], we conducted a detailed study of where time really goes during the evaluation of loop-lifted XQuery plans. For the unmodified instance of Query *Q11*, *e.g.*, almost half of the query evaluation time was dedicated to sort operations.

Queries *Q6* and *Q7* show speedups of several orders of magnitude (note the logarithmic scale above the gap in the y-axis). In Section 4.4.1, we briefly mentioned

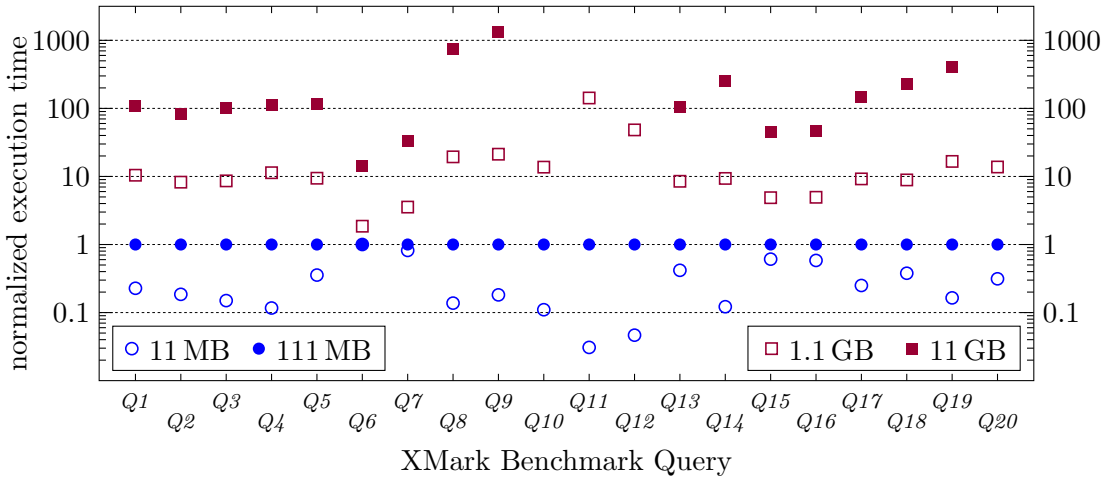


Figure 5.13: MonetDB/XQuery scalability with respect to document size. Execution times are normalized to the elapsed time on the 111 MB document instance.

that XPath step bundling may lead to additional rewrites due to now adjacent step operators in the algebraic plan (page 97). Such opportunities arise in the plans for Queries *Q6* and *Q7*. In the initial plan, a ϱ operator separated the two step operators $\text{⌐}_{\texttt{descendant-or-self::node()}}$ and $\text{⌐}_{\texttt{child::}\nu}$. After removing the operator, the now adjacent steps could be merged into $\text{⌐}_{\texttt{descendant::}\nu}$.

5.4.4 Scalability with Respect to Data Volumes

Our prime motivation to employ relational databases for XML data processing was their outstanding *scalability* in modern implementations. Obviously, we want to assess whether MonetDB/XQuery meets our expectations of being a scalable XQuery platform.

To this end, we normalized the execution times observed on documents from 11 MB to 11 GB size to the elapsed time on the 111 MB instance. If our system reaches linear scalability with respect to document size, we expect the observed execution times to increase by a factor of 10 whenever the underlying document size increases by the same factor. Ideally, the measurements in Figure 5.13 are expected to arrange along the horizontal dotted lines, each of which mark a tenfold increase in execution time.

For the majority of the benchmark queries this is in fact true. The only troublemakers are the two join queries *Q11* and *Q12*, both of which observe quadratic scaling with respect to document size. The cause for this behavior is quite easy to comprehend if we look into the query text of *Q11* and *Q12*.

Both of the queries make use of the *inequality* predicate

```
where $p/profile/@income > 5000 * $i/text()
```

to relate `person` elements to auction items they could afford. The result set of this relation is huge: 120 thousand to 120 billion tuples are produced for the join on the 11 MB and 11 GB document instances, respectively. Though the final outcome of this query is small (due to a subsequent aggregation with `fn:count ()`), a quadratic complexity is inherent to both queries. *Any* XQuery system is bound to exhibit this complexity for Queries *Q11* and *Q12*.

Queries *Q6*, *Q7*, *Q15*, and *Q16*, in contrast, even show a *sub-linear* scaling. We benefit from one of the strengths of relational database technology here. Queries *Q6* and *Q7* take advantage of efficient implementations for *aggregation* in MonetDB/XQuery (XQuery function `fn:count ()`). All four queries exploit the presence of *name indexes* in the system. They allow for the application of *name test pushdowns* which we already found effective for the performance of our MonetDB-based staircase join implementation in Chapter 3.

Queries *Q8–Q10* and *Q20* failed to meet the linear scalability goal only for the multi-gigabyte documents. This is due to an insufficient amount of swap space on the test machine we used and is not a problem inherent to our approach.

5.4.5 XQuery on High Data Volumes

Table 5.2 concludes our experimental assessment of MonetDB/XQuery with a lineup of the XMark execution times we measured for document sizes up to 11 GB. Even in the area of multi-gigabyte XML instances, we see response times that allow for interactive querying. MonetDB/XQuery clearly proves the viability of our purely relational approach. Only minimal modifications to the relational system itself were required to push the limits of high-volume XML processing beyond the gigabyte limit.

5.5 Research in the Neighborhood

A close look into the specifics of both, loop-lifted query evaluation plans and our source language XQuery, revealed a number of powerful leverage points for relational query optimization in Pathfinder. This turned the purely relational MonetDB/XQuery system into one of the fastest XQuery processors available today. Several approaches in the literature are related to the ideas we described here.

XMark	execution time [ms]						
Query	11 MB	34 MB	111 MB	335 MB	1.1 GB	3.3 GB	11 GB
Q1	15	23	66	201	688	2235	7046
Q2	23	42	124	318	1021	3164	10262
Q3	109	235	726	1929	6257	20377	73044
Q4	23	62	197	647	2234	7000	22128
Q5	16	21	45	124	424	1643	5216
Q6	7	7	7	8	13	29	101
Q7	9	9	11	17	39	101	367
Q8	45	88	327	1269	6354	63816	241688
Q9	63	108	346	1431	7351	77125	452255
Q10	318	838	2885	11057	39568	–	–
Q11	77	299	2505	19673	356421	–	–
Q12	59	170	1266	10299	61195	–	–
Q13	23	30	55	136	467	1451	5830
Q14	81	218	664	2095	6211	17906	170311
Q15	25	29	41	77	200	563	1866
Q16	28	32	48	89	237	673	2236
Q17	24	38	96	273	884	4184	14293
Q18	14	19	37	98	329	1162	8392
Q19	40	77	243	683	4038	9862	97651
Q20	72	101	230	990	3170	10070	–

Table 5.2: MonetDB/XQuery performance for XMark queries *Q1–Q20*.

5.5.1 Algebraic Optimization for XQuery

The availability of a query optimizer is crucial to the efficiency of any mature database management system. This is why most of the existing XML databases rely on algebraic means to rephrase queries such that they execute most efficiently. The Natix system by Moerkotte *et al.* [Fiebig02, Brantner05], the Galax system by Fernández *et al.* [Re06], and the Timber system by Jagadish *et al.* [Jagadish02, Jagadish01] are representatives of such systems.

All of these implementations, however, use algebra dialects that have specifically been crafted for the XML/XQuery domain. XQuery `FLWOR` expressions, for instance, have direct equivalents in the algebras used by Natix and Galax (*map* operators). Path expressions are translated into *tree patterns* in Galax and Timber. Quite in contrast to that, our approach is a purely relational one, without the need for explicit XQuery extensions. As such, well-proven techniques from existing systems are directly applicable to Pathfinder's optimizer and vice versa.

Relational Optimization

A comprehensive list of relational optimization techniques can be found in the survey article of Jarke and Koch [Jarke84]. The rewrite strategies suggested there include the pushdown of projection operators across "constructive" algebra operators (joins, Cartesian products), an idea for which we found an efficient and novel implementation for DAG-shaped plans. In turn, the pushdown of *selections* described by [Jarke84] is not yet generally addressed in the Pathfinder compiler. Its implementation might yield additional performance advantages in the future.

An important aspect of the plans we consider is the significant amount of *sharing* in the plans' *DAG representation*. But though a physical implementation of shared subplans in terms of the *split* operator has long since been proposed by Graefe [Graefe93], the problem of optimizing DAG-shaped plans still remains largely unaddressed. Only recently, the thesis of Neumann [Neumann05] took a close look into that matter, which would certainly facilitate the optimization of loop-lifted plan DAGs.

XQuery Join Detection

We have stressed the importance of a reliable *join detection* for XQuery processing. Yet, it seems quite surprising that until now no convincing approach has been published to recognize join situations in XQuery in a robust algebraic manner. Several existing implementations apparently do make use of joins for efficient XQuery evaluation (as described, *e.g.*, for FluX [Koch04] and System RX [Beyer05]). The detection of joins, however, seems primarily driven by syntactical analyses of the input query itself. As such, the detection tends to be highly fragile with respect to variations in the input query. In fact, in the experiments we performed in the course of [Boncz06b], we found existing systems only recognize few of the five join queries in the XMark benchmark set.

5.5.2 Order Awareness

The tight ordering constraints of the XQuery language manifest in the frequent use of expensive row numbering operators in loop-lifted XQuery evaluation plans. In return, the XQuery implementation MonetDB/XQuery can significantly benefit from the *lack* of explicit order requirements in (parts of) the query. We are not aware of any other XQuery implementation that exploits this optimization hook to the extent described here. In fact, with the exception of Saxon [Kay], we have found no traces of order indifference in other open-source XQuery engines. The built-in function `fn:unordered()`, *e.g.*, is commonly implemented as the identity function.

The Galax system incorporates a limited degree of order awareness in terms of the *duptidy* automaton described by Fernández *et al.* [Fernández04]. The automaton analyzes a sequence of path steps with respect to the guaranteed order properties of its result sequence. The inference, however, assumes a fixed (and rather naïve) evaluation strategy for each axis and is hardly extendible to further constructs in the XQuery language.

The inference of *physical* order properties in Pathfinder resembles the concept of *interesting orders* in the System R optimizer [Selinger79]. In addition, we introduced the less restrictive ordering criterion $\langle a_1, \ldots, a_n \| p \rangle$ to adequately exploit MonetDB's `HashRowNumber` operator. This is quite similar to the *secondary orderings* described by Wang and Cherniack [Wang03]. Neumann and Moerkotte [Neumann04] propose an order propagation framework that generalizes optimizations referring to ordering and grouping even further.

5.5.3 XQuery Cardinality Forecasts

Goldman and Widom [Goldman97] have already suggested the use of DataGuides as statistical summaries for XML documents, which we introduced into our compiler in terms of the statistical guide. Later work in this field mainly focused on minimizing the *space requirements* of such metadata—most notably *path trees*, *Markov tables* [Aboulnaga01], and *Bloom histograms* [Wang04]. These efforts take a rather small subset of XPath into account, namely *rooted paths* of *child steps* only. In contrast, our derivation process based on an algebraic query representation addresses cardinality forecasts for *arbitrary* XQuery expressions. We think that the accuracy of such forecasts easily outweighs the space requirements of the statistical guide. The experimental studies in [Goldman97] support our assumption that DataGuides typically remain small even for large XML documents.

The extraction of relevant paths in [Marian03] resembles our notion of *guide nodes*—though for a completely different purpose. In [Marian03], the proposed technique leads to a *projection* of XML documents at document loading time in order to minimize the runtime main memory requirements of Galax. Relevant paths are inferred by *simulating* the query at compile time. In contrast to our work, Galax simulates the evaluation of an XQuery Core expression itself (not an algebraic equivalent) and in absence of any schema information (such as our statistical guide).

5.5.4 Further Optimization Hooks

Others have suggested the inclusion of *schema information* into the process of optimizing XQuery expressions (*e.g.*, [Pal05, Koch04]). The exploitation of such information is orthogonal to the techniques described here. As such, schema-based

optimization may well be included into the optimization procedure of Pathfinder. The compiler's *schema import* facilities already provide for a significant share of the implementation needs required for that task.

6 Wrap-Up

Relational database systems constitute one of the best understood and engineered data management systems available today. Their core data model, *tables of tuples*, is simple and thus efficient to implement. The availability of *indexes* allows for a particularly fast execution of the predominant operations on tables: searching and scanning. The same systems can act as highly efficient *tree processors* if we carefully exploit the strengths of relational database technology for that purpose. *Tuples*, *e.g.*, may take the role of *nodes*, *index scans* suitably implement *tree navigation*, whereas *joins* imitate the *iteration* over sequences.

The need for such tree processors accumulates in the search for efficient implementations of *XQuery*, the emerging query language for XML document trees. In this thesis, we have confirmed the viability of a purely relational XQuery implementation. The *MonetDB/XQuery* system that is the outcome of this work is, in fact, among the fastest XQuery processors available today. We will now summarize the contributions that led to this successful implementation, all embodied in the system's XQuery compiler *Pathfinder*.

6.1 Summary

The outline of this thesis was inspired by the relational XQuery processing stack shown in Figure 6.1. Three key contributions leveraged the maturity of relational database systems into the domain of XQuery:

(i) the *XPath accelerator* tree encoding provides the isomorphism between the

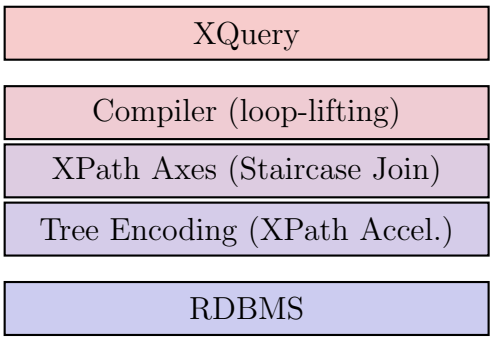


Figure 6.1: Relational XQuery Processing Stack.

relational data model and XML,

(ii) a novel algebra operator, *staircase join*, implements XPath navigation on such encoded data at unprecedented speed, and finally

(iii) the *loop-lifting* technique compiles arbitrary XQuery expressions into a purely relational algebra dialect.

A novel approach to relational *query optimization* completes the system to meet the desired performance goal.

6.1.1 Relational Tree Encodings

The foundation of our work is the *XPath accelerator* tree encoding proposed by Grust [Grust02]. In this encoding, each node's pre- and postorder ranks serve as a highly efficient mapping of the XML tree structure to tuples in a relational table doc. XPath location steps from arbitrary context nodes then translate into simple *range queries* over doc, efficiently implementable with existing indexing techniques.

Range Encoding as an Alternative. The XML tree structure *couples* the fundamental node properties *pre*, *post*, *size*, and *level*. As a consequence, the XPath accelerator encoding may easily be rephrased in terms of property sets equivalent to *pre*/*post*. Though equal to its predecessor at first sight, we proved that the *range encoding* (*pre*/*size*/*level*) slightly outperforms XPath accelerator for recursive XPath axes. Furthermore, the novel encoding simplifies the relational implementation of updates and node construction, which makes it our preferred relational storage means used in MonetDB/XQuery.

Partitioned B-Trees for Efficient Step Evaluation. Range-encoded tree data are quite a perfect fit for the concept of *partitioned B-trees* described by Graefe [Graefe03]. The use of column *level* as low-selectivity prefix of a partitioned B-tree, *e.g.*, compensates for the omission of an explicit *parent* pointer in the relational schema of the range encoding. Name and kind tests become highly efficient if the index is prefixed with the *kind* and/or *prop* fields.

Single Record Scans for Early-Out Semantics. The use of *single record* scans in relational query plans elegantly mirrors XQuery's *early-out* semantics. We found the same concept to provide an efficient implementation of `parent` steps on range-encoded tree data. The tools required for this optimized evaluation strategy are readily available in existing implementations.

A thorough experimental study confirmed the efficiency of the tree encoding for relational XPath evaluation.

6.1.2 XPath Evaluation with Staircase Join

We have presented a novel join operator, *staircase join*, that provides outstanding XPath performance on encoded tree data. Staircase join encapsulates comprehensive knowledge on the data's underlying tree structure in a single operator. As such, it may be easily plugged into any RDBMS kernel to efficiently back the evaluation of XPath location steps. The effectiveness of staircase join is due to three specific techniques incorporated in the operator: *pruning*, *partitioning*, and *skipping*.

Pruning. The context sequence of an XPath location step may contain nodes that will not directly contribute to the step result due to their overlapping query regions. Staircase join will *remove* such nodes from the context before the actual join processing, while fully retaining the possibility of a *pipelined* join execution.

Partitioning. Even after pruning, partially overlapping regions may still lead to the production of duplicate result nodes. By *partitioning* the *pre/post* plane using the preorder ranks of the remaining context nodes, staircase join ensures a duplicate-free result sequence, sorted in document order.

Skipping. Based on a careful examination of the tree origin of the encoded data, we concluded the *emptiness* of certain regions in the *pre/post* plane. With this knowledge in mind, we tuned staircase join to *skip* over the unpopulated space during join processing. The effectiveness of this technique is high. On actual

query workloads, we found staircase join save more than 90 % of processing work by skipping empty *pre/post* regions.

Any relational database may benefit from staircase join to speed up XPath evaluation. To prove this claim, we implemented staircase join in the disk-based PostgreSQL system as well as in the main memory database kernel MonetDB. The outcome was an outstanding XPath performance on both systems.

6.1.3 Loop-Lifting: A Relational Approach to Iteration

To bridge the semantical gap between the *set-oriented* processing model of relational databases and XQuery which operates on *sequences* of items, we introduced *loop-lifting*, a novel compilation approach to XQuery. The semantical correctness of this compilation procedure is ensured by our careful choice of a relational representation of *sequences* and *iteration*.

Loop-Lifted `FLWOR` Compilation. The functional-style semantics of iteration in `FLWOR` clauses allows for the independent (or parallel) evaluation of the body of an XQuery `for` clause. We expressed this independence in terms of the *iter* column in our relational sequence encoding, which allows for a fully *set-oriented* evaluation of the `for` iteration primitive. The same column captures XQuery's concept of *iteration order*, while we implemented *sequence order* in terms of the column *pos* of the $\langle iter, pos, item \rangle$ sequence encoding.

Efficient Handling of Live Node Sets. As part of the relational translation process, our compiler infers additional information about the origin of the XML tree nodes in the result. The exploitation of this *live node set* information led to a significant speedup in query execution times as observed in experiments on IBM DB2.

Relational Implementation of Dynamic Typing in XQuery. The compilation procedure we devise covers the relational implementation of runtime tests on XQuery *types*, a feature that was even considered *untranslatable* for purely relational back-ends in the past [Manolescu01]. Our implementation is based on *aggregation*, a feature that existing RDBMSs know well how to evaluate efficiently.

The loop-lifted compilation makes XQuery accessible to *any* relational system. To demonstrate, we ran a subset of the XMark benchmark set on the SQL system DB2, a setup that readily provides scalability to gigabyte-sized XML instances.

6.1.4 Query Optimization for Loop-Lifted XQuery Plans

The loop-lifted compilation of XQuery results in query plans whose shape is quite different from the usual $\pi-\sigma-\bowtie$ pattern typical for relational systems. A novel set of query optimization techniques accounts for that fact. Though initially designed for loop-lifted XQuery evaluation plans, the techniques are universal by nature and may prove efficient in other contexts as well.

Peephole-Style Plan Analysis. The sheer size of loop-lifted query plans defeats the use of a classical pattern-driven optimization process. In a single plan traversal, the query optimizer in Pathfinder infers a set of relevant *plan annotations* to prepare for the analysis of the plan. Plan nodes may then be looked at from a *peephole* view to guide an efficient and scalable rewrite process.

Order-Aware Optimization. Frequent occurrences of the relational row numbering operator ϱ make the prevalence of *order* in XQuery shine through in loop-lifted XQuery evaluation plans. The Pathfinder compiler carefully considers order constraints (including the lack thereof) during its optimization phase. To our knowledge, we presented the first XQuery implementation that actually draws advantage from the *indifference* of order in XQuery's `unordered{·}` and `fn:unordered()` clauses.

Reliable Cardinality Forecasts for XQuery. In preparation of a reliable cost model for loop-lifted XQuery evaluation plans, we have introduced a dependable means to derive *result size estimates* for arbitrary XQuery expressions. The derivation process depends on the availability of a *statistical guide* and a new notion of *guide nodes*. In contrast to existing approaches, which provide estimates only for a simple subset of XPath expressions, the technique we have presented applies to XQuery expressions of arbitrary shape.

6.1.5 MonetDB/XQuery: The Proof of Our Claim

We have shown that relational databases may back the evaluation of XQuery in a standards-compliant manner. To prove our claim that the approach is also viable in practice, we constructed a complete XQuery implementation on the basis of the MonetDB RDBMS kernel. The system is available now, both, for real use and as a research and experimentation platform under an open-source license.[1]

In Chapter 5, we used this system for a detailed study of the performance of loop-lifted XQuery evaluation. We confirmed an unprecedented scalability with

[1] `http://www.monetdb-xquery.org/`

	110 MB				1.1 GB			11 GB
Query	MXQ	Galax	XHive	BDB	MXQ	XHive	BDB	MXQ
Q1	0.12	0.72	1.29	0.51	1.3	9.9	5.9	14
Q2	0.19	0.31	1.75	1.38	1.8	33.0	43.1	19
Q3	1.20	1.76	5.66	3.55	11.5	25.1	37.1	176
Q4	0.42	2.91	1.00	4.07	4.5	18.1	43.3	44
Q5	0.08	0.63	0.90	1.05	0.8	20.7	11.4	10
Q6	0.00	13.29	10.17	13.23	0.0	178.1	–	0.1
Q7	0.01	30.01	24.84	14.70	0.1	278.4	–	0.6
Q8	0.47	2.12	3.51	9316.72	9.6	49.1	–	223
Q9	0.52	–	12280.66	–	11.8	–	–	460
Q10	5.18	18.61	442.37	–	62.8	–	–	2413
Q11	3.62	–	19927.29	–	367.7	–	–	–
Q12	2.11	–	5100.19	–	121.1	–	–	–
Q13	0.10	0.66	1.03	0.79	0.9	12.9	8.1	8
Q14	0.93	99.53	11.16	14.18	7.5	110.2	–	452
Q15	0.07	0.20	0.49	1.37	0.4	10.6	28.5	3
Q16	0.08	0.46	0.52	1.52	0.5	10.9	17.6	4
Q17	0.15	0.82	0.85	2.08	1.4	11.8	34.1	31
Q18	0.05	0.73	0.64	2.09	0.5	14.8	21.7	7
Q19	0.38	14.73	12.15	6.74	7.0	254.5	135.6	128
Q20	0.62	2.98	1.40	3.42	7.0	24.6	37.4	70

Table 6.1: XMark query performance (elapsed time in seconds) of four different XQuery implementations: MonetDB/XQuery (MXQ), Galax, X-Hive, and Berkeley DB XML (BDB). Results obtained in the course of our work in [Boncz06b].

interactive response times up to and beyond the multi-gigabyte XML document range. Our experiments reflect the effectiveness of the relational XQuery evaluation techniques we have described in this thesis.

In [Boncz06b], we conducted a comparative assessment of MonetDB/XQuery and three other XQuery systems currently available: *(i)* Galax 0.5.0 [Fernández03], *(ii)* X-Hive/DB 6.0 [X-Hive/DB], and *(iii)* Berkeley DB XML 2.2 [BDB]. They all had to compete against version 0.10.2 of MonetDB/XQuery on a 1.6 GHz AMD Opteron system with 8 GB RAM. In Table 6.1, we lined up the query execution times observed for document sizes ranging from 110 MB to 11 GB (XMark scale factors 1 to 100). In this setup, MonetDB/XQuery clearly outclasses its competitors and reinforces its role as one of the fastest XQuery implementations currently in existence.

6.2 Ongoing and Future Work

We have developed a complete stack of novel techniques that allow for XQuery evaluation by highly scalable relational means. Yet, this thesis marks the beginning rather than the end of research on relational XQuery processing. In a joint effort, the research groups at the Technische Universität München, the University of Twente, and the CWI Amsterdam are actively pursuing the further development of the MonetDB/XQuery system and its query compiler Pathfinder.

6.2.1 Alternative Back-Ends for Pathfinder

The Pathfinder compiler has been designed in a platform-independent manner, with an intermediate algebra representation that makes only minimal assumptions about the underlying back-end. In the future, we plan to extend Pathfinder's support to database systems other than MonetDB. From a research perspective, two major challenges will arise in this context.

Pipelined Execution. MonetDB performs full *materialization* of intermediate query results, whereas other back-ends will typically strive for a *pipelined* execution. To deal with DAG-shaped plans, a pipelining back-end will have to provide a *split* operator that caches its tuple stream if two consumers read the data at a different pace. Since the difference in reading pace determines the degree of materialization required to implement *split*, we strive for an *equalization* of the consumption rate among all parents of a *split* operator to ensure maximum performance.

Advanced Index Usage. For lack of B-tree support in the MonetDB system, many of the efficient access techniques discussed in Chapters 2 and 3 are not readily accessible in MonetDB/XQuery. Hence, their exploitation has not yet been addressed in the Pathfinder compiler. The incorporation of additional XPath evaluation techniques (*e.g.*, [Bruno02, Al-Khalifa02]) may add another challenge.

6.2.2 Further Optimization Hooks

Though quite effective in the implementation of MonetDB/XQuery, the optimizer of Pathfinder is still in an early stage. A number of optimization opportunities are sitting in wait to further improve the system's performance.

Cost Models for Loop-Lifted Plan DAGs. Probably the largest shortcoming of the current system is the provisional implementation of its *cost model*. One of the

building blocks for the re-implementation of this component will be the cardinality derivation procedure we have described in Chapter 5. We are currently developing the missing pieces for an accurate cost model in Pathfinder based on the findings of Manegold [Manegold02].

Schema-Based Algebra Optimization. The MonetDB/XQuery system carries out efficient XML processing in a purely schema-oblivious fashion. The drawback of this approach is that the system does not benefit from additional optimization hooks that arise from the presence of *schema information*. The examination of such information could trigger advanced rewrite rules in the query optimizer of Pathfinder.

6.2.3 Exploring New Fields of Knowledge

The Pathfinder system has long since surpassed mere XQuery support. Since its first public release, Pathfinder has indeed found its way into unexpected fields of knowledge, such as a distributed peer-to-peer query processing system [Zhang05], the XIRAF system developed at the Dutch Forensic Institute [Alink05], and the Tijah text retrieval system in the course of the MultimediaN project [Flokstra].

Acknowledgments

The cornerstone for Pathfinder was laid about five years ago in Konstanz, when Torsten Grust and I created its first source file: `XQuery.y`. Since then, Torsten has coached me through the very effective and fruitful development of Pathfinder, for which I'd like to express my sincere thanks. We had never anticipated at that time, though, that our little piece of software would soon grow into a system that would start to spread all over the world, finding its way into research as well as industry projects.

Actually, our original intent of the Pathfinder software was to build a very low-footprint XQuery implementation, meant for embedded devices and handheld PCs. However, we realized quite early that our strengths rather lie in the field of large-scale databases and the idea of relational XQuery processing began to grow. Ironically, Pathfinder is currently making its way back into embedded devices: in an industry-sponsored project at the CWI, MonetDB/XQuery has successfully been ported to be used in electronic entertainment devices.

About a year after I had started to work on Pathfinder, Maurice van Keulen joined the team when he was in Konstanz for a one-year sabbatical. The outcome of this joint effort was not only the development of staircase join. What is more, Maurice also spread the word on Pathfinder to the Netherlands and brought us in contact with the folks from the CWI, whom we first met at VLDB 2003 in Berlin. Thanks Maurice!

Headed by Martin Kersten—whom I am also grateful for volunteering to review my thesis—, the MonetDB group pushed (in a positive sense!) and supported us to turn MonetDB/XQuery into what it is today. Most notable in this context are Peter Boncz and Stefan Manegold who spent a lot of effort to make our software a success.

At the same time, Jan Rittinger got more and more involved in the project as a student. Not only his Bachelor and Master's theses contributed significantly to Pathfinder. In the summer of 2004, he spent a half-year internship at the CWI in Amsterdam, where he implemented the final pieces to get the MonetDB/XQuery system up and running. The file that hosts most of the code from that time, `milprint_summer.c`, has actually become a synonym for a whole development era in the Pathfinder project.

There are numerous other people, without whom this thesis would not have been possible. Most of my work on Pathfinder was done at the University of Konstanz, where my supervisor Marc Scholl gave me all the freedom and support I could imagine. Several students from Konstanz contributed directly or indirectly to this work.

Last but not least, I would like to dedicate this thesis to my little daughter Mia and thank my wife Sabine and my family for their constant support.

Bibliography

[Aboulnaga01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, pages 591–600. Rome, Italy, September 2001.

[Al-Khalifa02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of the 18th Int'L Conference on Data Engineering (ICDE)*. San Jose, CA, USA, February 2002.

[Alink05] Wouter Alink. *XIRAF—an XML Information Retrieval Approach to Digital Forensics*. Master's thesis, University of Twente, 2005.

[Amagasa03] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. QRS: A Robust Numbering Scheme for XML Documents. In *Proc. of the 19th Int'l Conference on Data Engineering (ICDE)*, pages 705–707. Bangalore, India, March 2003.

[Antimirov95] Valentin M. Antimirov. Rewriting Regular Inequalities. In *Proc. of the 10th Int'l Symposium on Fundamentals of Computation Theory (FCT)*, pages 116–125. Dresden, Germany, August 1995.

[Bayer77] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Transactions on Database Systems (TODS)*, 2(1), pages 11–26, March 1977.

[BDB] Berkeley DB XML. `http://www.sleepycat.com/products/bdbxml.html`.

[Beyer05] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: One Part Relational, One Part XML. In *Proc. of the 2005 ACM SIGMOD Int'l Conference on Management of Data*, pages 347–358. Baltimore, MD, USA, June 2005.

[Boag05] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. World Wide Web Consortium Candidate Recommendation, September 2005. `http://www.w3.org/TR/xquery/`.

[Boncz99] Peter A. Boncz and Martin L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2), pages 101–119, October 1999.

[Boncz02] Peter A. Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. Ph.D. thesis, Universiteit van Amsterdam, May 2002.

[Boncz05a] Peter Boncz, Torsten Grust, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: Relational XQuery over Multi-Gigabyte XML Inputs in Interactive Time. Technical Report INS-E0503, CWI, Amsterdam, March 2005.

[Boncz05b] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Loop-Lifted Staircase Join: From XPath to XQuery. Technical Report INS-E0510, CWI, Amsterdam, March 2005.

[Boncz05c] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: XQuery—The Relational Way. In *Proc. of the 31st Int'l Conference on Very Large Databases (VLDB)*, pages 1322–1325. Trondheim, Norway, September 2005.

[Boncz05d] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. of the 2nd Int'l Conference on Innovative Data Systems Research (CIDR)*, pages 225–237. Asilomar, CA, USA, January 2005.

[Boncz06a] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Sjoerd Mullender, Jan Rittinger, and Jens Teubner. MonetDB/XQuery–Consistent & Efficient Updates on the Pre/Post Plane. In *Proc. of the 10th Int'l Conference on Extending Database Technology (EDBT)*, pages 1190–1193. Munich, Germany, March 2006.

[Boncz06b] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of the 2006 ACM SIGMOD Int'l Conference on Management of Data*. Chicago, IL, USA, June 2006.

[Brantner05] Matthias Brantner, Carl-Christian Kanne, Sven Helmer, and Guido Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proc. of the*

21st Int'L Conference on Data Engineering (ICDE), pages 705–716. Tokyo, Japan, April 2005.

[Bruno02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, pages 310–321. Madison, WI, USA, 2002.

[Chamberlin06] Don Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery Update Facility. World Wide Web Consortium Working Draft, January 2006. `http://www.w3.org/TR/xqupdate/`.

[Chan04] Chee-Yong Chan and Wenfei Fan. Taming XPath Queries by Minimizing Wildcard Steps. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 156–167. Toronto, Canada, September 2004.

[Cohen02] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML Trees. In *Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 271–281. Madison, WI, USA, June 2002.

[DeHaan03] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. of the 2003 ACM SIGMOD Int'l Conference on Management of Data*, pages 623–634. San Diego, CA, USA, June 2003.

[Deutsch99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *Proc. of the 1999 ACM SIGMOD Int'l Conference on Management of Data*, pages 431–442. Philadelphia, PA, USA, June 1999.

[Draper05] Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium Candidate Recommendation, September 2005. `http://www.w3.org/TR/xquery-semantics/`.

[Fernández03] Mary F. Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax Experience. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, pages 1077–1080. Berlin, Germany, September 2003.

[Fernández04] Mary Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Automata for Avoiding Unneccessary Ordering Operations in XPath Evaluation Plans. Technical Report UA 04-02, University of Antwerp, 2004.

[Fernández05] Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model. World Wide Web Consortium Candidate Recommendation, September 2005. `http://www.w3.org/TR/xpath-datamodel/`.

[Fiebig02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, and Robert Schiele. Anatomy of a native XML base management system. *The VLDB Journal*, 11(4), pages 292–314, December 2002.

[Finkel74] Raphael A. Finkel and Jon Louis Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4, pages 1–9, 1974.

[Flokstra] Jan Flokstra, Henning Rode, Djoerd van Hiemstra, and Roel van Os. MultimediaN—Semantic Multimedia Access. `http://www.multimedian.nl/`.

[Florescu99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), pages 27–34, September 1999.

[Gluche97] Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental Updates for Materialized OQL Views. In *Proc. of the 5th Int'l Conference on Deductive and Object-Oriented Databases (DOOD'97)*, pages 52–66. Montreux, Switzerland, December 1997.

[Goldman97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization. In *Proc. of the 23rd Int'l Conference on Very Large Databases (VLDB)*, pages 436–445. Athens, Greece, August 1997.

[Gottlob05] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems (TODS)*, 30(2), pages 444–491, June 2005.

[Graefe93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), pages 73–170, June 1993.

[Graefe03] Goetz Graefe. Sorting and Indexing with Partitioned B-Trees. In *Proc. of the 1st Int'l Conference on Innovative Data Systems Research (CIDR)*. Asilomar, CA, USA, January 2003.

[Grün06] Christian Grün, Alexander Holupirek, Marc Kramis, Marc H. Scholl, and Marcel Waldvogel. Pushing XPath Accelerator to its Limits, 2006. (Under submission).

[Grust02] Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, pages 109–120. Madison, WI, USA, June 2002.

[Grust03a] Torsten Grust and Maurice van Keulen. Tree Awareness for Relational DBMS Kernels: Staircase Join. In Henk Blanken, Torsten Grabs, Hans-Jörg Schek, Ralf Schenkel, and Gerhard Weikum (editors), *Intelligent Search on XML Data*, Lecture Notes in Computer Science. Springer Verlag, September 2003.

[Grust03b] Torsten Grust, Maurice van Keulen, and Jens Teubner. Bridging the Gap Between Relational and Native XML Storage with Staircase Join. In *Proc. of the 15th GI Workshop on Foundations of Database Systems*, pages 85–89. Tangermünde, Germany, June 2003.

[Grust03c] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, pages 524–535. Berlin, Germany, September 2003.

[Grust04a] Torsten Grust, Jan Hidders, Philippe Michiels, Roel Vercammen, and Maurice van Keulen. Supporting Positional Predicates in Efficient XPath Axis Evaluation for DOM Data Structures. Technical Report UA 2004-05, University of Antwerp, 2004.

[Grust04b] Torsten Grust and Stefan Klinger. Schema Validation and Type Annotation for Encoded Trees. In *Proc. of the ACM SIGMOD/PODS 1st Int'l Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*. Paris, France, June 2004.

[Grust04c] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 252–263. Toronto, Canada, September 2004.

[Grust04d] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Proc. of the 1st Twente Data Management Workshop (TDM)*, pages 7–14. Enschede, The Netherlands, June 2004.

[Grust04e] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems (TODS)*, 29(1), pages 91–131, March 2004.

[Grust05] Torsten Grust. Purely Relational FLWORs. In *Proc. of the ACM SIGMOD/PODS 2nd Int'l Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*. Maryland, MD, USA, June 2005.

[Grust06] Torsten Grust, Jan Rittinger, and Jens Teubner. eXrQuy: Order Indifference in XQuery, 2006. (Under submission).

[Guttman84] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the 1984 ACM SIGMOD Int'l Conference on Management of Data*, pages 47–57. Boston, MA, USA, June 1984.

[Helmer02] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Optimized Translation of XPath into Algebraic Expressions. In *Proc. of the 3rd Int'l Conference on Web Information Systems Engineering (WISE)*, pages 215–224. IEEE Computer Society, Singapore, December 2002.

[Hosoya05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1), pages 46–90, March 2005.

[Int05] Intel Corporation. *IA-32 Intel® Architecture Optimization Reference Manual*, June 2005.

[Jagadish01] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *Database Programming Languages (DBPL), 8th Int'l Workshop*, pages 149–164. Frascati, Italy, September 2001.

[Jagadish02] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4), pages 274–291, December 2002.

[Jarke84] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2), pages 111–152, June 1984.

[Kay] Michael Kay. The Saxon XSLT and XQuery Processor. `http://saxon.sf.net/`.

[Kempa03] Martin Kempa and Volker Linnemann. Type Checking in XOBE. In *Proc. of the 2003 BTW Conference (Datenbanksysteme für Business, Technologie und Web)*, pages 227–246. Leipzig, Germany, February 2003.

[Kepser04] Stephan Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Proc. of the Extreme Markup Languages 2004*. Montréal, Quebec, Canada, August 2004.

[Kersten05] Martin L. Kersten and Stefan Manegold. Cracking the Database Store. In *Proc. of the 2nd Int'l Conference on Innovative Data Systems Research (CIDR)*, pages 213–224. Asilomar, CA, USA, January 2005.

[Kha02] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. A Structural Numbering Scheme for XML Data. In *XML-Based Data Management and Multimedia Engineering—EDBT 2002 Workshops*, pages 91–108. Prague, Czech Republic, March 2002.

[Koch04] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*. Toronto, Canada, September 2004.

[Krishnamurthy03] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F. Naughton. XML–SQL Query Translation Literature: The State of the Art and Open Problems. In *Proc. of the 1st Int'l XML Database Symposium (XSym)*, pages 1–18. Berlin, Germany, September 2003.

[Lee96] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. Index Structures for Structured Documents. In *Proc. of the 1st Int'l Conference on Digital Libraries (DL)*, pages 91–99. Bethesda, MD, USA, 1996.

[Ley] Michael Ley. Computer Science Bibliography. `http://dblp.uni-trier.de/`.

[Li01] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, pages 361–370. Rome, Italy, September 2001.

[Makins95] Marian Makins (editor). *Collins English dictionary*. HarperCollins Publishers, Glasgow, UK, 3rd edition, 1995. ISBN 0 00 470677-3.

[Manegold02] Stefan Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. Ph.D. thesis, Universiteit van Amsterdam, December 2002.

[Manolescu01] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, pages 241–250. Rome, Italy, September 2001.

[Marian03] Amélie Marian and Jérôme Siméon. Projecting XML Documents. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, pages 213–224. Berlin, Germany, September 2003.

[Mayer04a] Sabine Mayer. *Enhancing the Tree Awareness of a Relational DBMS: Adding Staircase Join to PostgreSQL*. Master's thesis, University of Konstanz, February 2004. `http://www.ub.uni-konstanz.de/kops/volltexte/2004/1166/`.

[Mayer04b] Sabine Mayer, Torsten Grust, Maurice van Keulen, and Jens Teubner. An Injection with Tree Awareness: Adding Staircase Join to PostgreSQL. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 1305–1308. Toronto, Canada, September 2004.

[McHugh99] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. of the 25th Int'l Conference on Very Large Databases (VLDB)*, pages 315–326. Edinburgh, Scotland, UK, September 1999.

[McKeeman65] William M. McKeeman. Peephole Optimization. *Communications of the ACM*, 8(7), pages 443–444, July 1965.

[Meijer05] Erik Meijer and Brian Beckman. XLinq: XML Programming Refactored (The Return Of The Monoids). In *Proc. of the 2005 XML Conference & Exposition*. Atlanta, GA, USA, November 2005.

[Melton03] Jim Melton. *Advanced SQL:1999: Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann Publishers, Amsterdam, 2003. ISBN 1-55860-677-7.

[Neumann04] Thomas Neumann and Guido Moerkotte. A Combined Framework for Grouping and Order Optimization. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 960–971. Toronto, Canada, September 2004.

[Neumann05] Thomas Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. Ph.D. thesis, Universität Mannheim, July 2005.

[Nicola05] Matthias Nicola and Bert van der Linden. Native XML Support in DB2 Universal Database. In *Proc. of the 31st Int'l Conference on Very Large Databases (VLDB)*, pages 1164–1174. Trondheim, Norway, September 2005.

[Nievergelt84] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems (TODS)*, 9(1), pages 38–71, 1984.

[Olteanu02] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *XML-Based Data Management and Multimedia Engineering, EDBT 2002 Workshops, Revised Papers*, pages 109–127. Prague, Czech Republic, March 2002.

[O'Neil04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of the 2004 ACM SIGMOD Int'l Conference on Management of Data*, pages 903–908. Paris, France, June 2004.

[Page05] Wim Le Page, Jan Hidders, Jan Paredaens, Roel Vercammen, and Philippe Michiels. On the Expressive Power of Node Construction in XQuery. In *Proc. of the 8th Int'l Workshop on the Web and Databases (WebDB 2005)*, pages 85–90. Baltimore, MD, USA, June 2005.

[Pal04] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML Data Stored in a Relational Database. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 1134–1145. Toronto, Canada, September 2004.

[Pal05] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, and Eugene Kogan. XQuery Implementation in a Relational Database System. In *Proc. of the 31st Int'l Conference on Very Large Databases (VLDB)*, pages 1175–1186. Trondheim, Norway, September 2005.

[Pal06] Shankar Pal, Dragan Tomic, Brandon Berg, and Joe Xavier. Managing Collections of XML Schemas in Microsoft SQL Server 2005. In *Proc. of the 10th Int'l Conference on Extending Database Technology (EDBT)*, pages 1102–1105. Munich, Germany, March 2006.

[PIR] Georgetown University Medical Center PIR. Integrated Protein Classification Database. `http://pir.georgetown.edu/iproclass/`.

[PostgreSQL] PostgreSQL. `http://www.postgresql.org/`.

[Ramakrishnan95] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Deductive Database Systems. *Journal of Logic Programming*, 23(2), pages 125–149, May 1995.

[Re06] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. of the 22nd Int'L Conference on Data Engineering (ICDE)*. Atlanta, GA, USA, April 2006.

[Rode03] Henning Rode. *Methods and Cost Models for XPath Query Processing in Main Memory Databases*. Master's thesis, University of Konstanz, October 2003. `http://www.ub.uni-konstanz.de/kops/volltexte/2004/1195/`.

[Ross02] Kenneth A. Ross. Conjunctive Selection Conditions in Main Memory. In *Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 109–120. Madison, WI, USA, June 2002.

[Sakr06] Sherif Sakr, Jens Teubner, and Torsten Grust. Dependable Cardinality Forecasts in an Algebraic XQuery Compiler, 2006. (In preparation).

[SAX] SAX—Simple API for XML. `http://www.saxproject.org/`.

[Schmidt00] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient Relational Storage and Retrieval of XML Documents. In *The World Wide Web and Databases (WebDB), 3rd Int'l Workshop*, pages 137–150. Dallas, TX, USA, May 2000.

[Schmidt02] Albrecht R. Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB)*, pages 974–985. Hong Kong, China, August 2002.

[Selinger79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the 1979 ACM SIGMOD Int'l Conference on Management of Data*, pages 23–34. Boston, MA, USA, 1979.

[Shanmugasundaram99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of the 25th Int'l Conference on Very Large Databases (VLDB)*, pages 302–314. Morgan Kaufmann, Edinburgh, Scotland, UK, September 1999.

[SQL06] Microsoft Corporation. *SQL Server Language Reference: Transact-SQL*, March 2006.

[Tatarinov02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered

XML Using a Relational Database System. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, pages 204–215. Madison, WI, USA, 2002.

[Wadler90] Philip Wadler. Comprehending Monads. In *Proc. of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78. Nice, France, June 1990.

[Wang03] Xiaoyu Wang and Mitch Cherniack. Avoiding Sorting and Grouping in Processing Queries. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, pages 826–837. Berlin, Germany, September 2003.

[Wang04] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 240–251. Toronto, Canada, September 2004.

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

[Yergeau03] Y. Yergeau. UTF-8, a Transformation Format of ISO 10646, November 2003. `http://www.ietf.org/rfc/rfc3629.txt`, request for Comments (RFC) 3629.

[Zhang01] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the 2001 ACM SIGMOD Int'l Conference on Management of Data*, pages 425–436. Santa Barbara, CA, USA, 2001.

[Zhang05] Jennie Zhang. P2P Query Processing on Top of MonetDB/XQuery. In *Dutch-Belgian Database Day (DBDBD)*. October 2005.