# Constructing a Relational Query Optimizer for Non-Relational Languages

**Dissertation**
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Jan Rittinger
aus Karlsruhe

Tübingen
2010

|  |  |
|---|---|
| Tag der mündlichen Qualifikation: | 08.04.2011 |
| Dekan: | Prof. Dr. Wolfgang Rosenstiel |
| 1. Berichterstatter: | Prof. Dr. Torsten Grust |
| 2. Berichterstatter: | Prof. Dr. Guido Moerkotte |

# Zusammenfassung

Die Speicherung von Daten in flachen, ungeordneten Tabellen sowie eine deklarative Anfragesprache sind Gründe für den Erfolg relationaler Datenbanksysteme: Sie erlauben einem Datenbankoptimierer nicht nur die Auswahl verschiedener Algorithmen, sondern auch die Wahl der besten Auswertungsreihenfolge. Dank jahrzehntelanger Forschung und Entwicklung zählen relationale Datenbanksysteme zu den besten Auswertungsplattformen für große Datenmengen.

In den meisten Programmiersprachen werden, im Gegensatz zu Datenbanksystemen, sortierte und mitunter verschachtelte Datenstrukturen verwendet. Die meisten Software-Entwickler arbeiten täglich mit diesen Datenstrukturen in der Programmiersprache ihrer Wahl, was dazu führt, dass das Schreiben von Datenbankanfragen oft ein Umdenken erfordert, beziehungsweise eine potentielle Fehlerquelle darstellt. Um die Vorteile von relationalen Datenbanksystemen für Entwickler in ihrer bekannten Umgebung nutzbar zu machen, stellen wir eine nicht-relationale Sprache vor, die mit Ordnung, verschachtelten Listen und komplexeren Datenstrukturen, wie zum Beispiel Tupeln, benannten Records und XML-Daten, umgehen kann. Wir übersetzen in dieser Sprache formulierte Anfragen in ungeordnete relationale Anfragen, die auf Tabellen arbeiten.

Die Übersetzung ist integraler Bestandteil des Compilers *Pathfinder* und beruht auf dem Konzept des *loop lifting*: Sie stellt die genaue Transformation der “fremden” Sprachkonzepte in die relationale Welt sicher. Die Zielsprache ist eine ungeordnete logische Algebra, deren Operatoren aus bekannten Algebra-Operatoren der Datenbankliteratur sowie zusätzlichen Nummerierungs- und XML-Operatoren besteht. Die zusätzlichen Operatoren ermöglichen die genaue Übersetzung von Ordnung, verschachtelten Listen und komplexeren Datenstrukturen.

Im Gegensatz zu normalen Datenbankanfragen besteht ein Algebraplan durchschnittlich nicht aus dutzenden, sondern aus hunderten von Operatoren. Die Kombination aus Größe des Plans, Vernetzung der Operatoren und Nummerierungsoperatoren überfordert alle von uns getesten Datenbankoptimierer. In dieser Arbeit stellen wir einen eigenen Optimierer vor, der die logischen Algebrapläne analysiert und jeden Operator mit Annotationen versieht. Diese Anmerkungen beschreiben Eigenschaften des umgebenden Planes und werden verwendet um gezielt lokale Plantransformationen vorzunehmen. Die Optimierungen werden durch Heuristiken gesteuert und führen jeweils zu einer inkrementellen Verbesserung des Plans. Ziel der Optimierungen ist das Entfernen möglichst vieler Operatoren—im Speziellen der Nummerierungsoperatoren—unter Berücksichtigung semantischer Äquivalenz.

Die vereinfachten Anfragepläne werden entweder in SQL-Anfragen oder in Skripte für das Hauptspeicher-Datenbanksystem MonetDB/XQuery übersetzt. MonetDB/XQuery kann, dank vieler XML-spezifischer Algorithmen und Erweiterungen, Anfragen auf XML-Daten sehr effizient auswerten. Die generierten SQL-Anfragen können stattdessen auf fast jedem relationalen Datenbanksystem ausgewertet werden und profitieren zusätzlich von den eingebauten Anfrageoptimierern der Datenbanksysteme. In unseren Experimenten analysieren wir die Qualität der optimierten Anfragepläne und vergleichen die Auswertungszeiten. Die untersuchten, optimierten Anfragen erinnern in ihrer Effizienz an handgeschriebene Anfragen.

Die Kombination aus Übersetzung in logische Algebrapläne, Optimierung und Generierung von Datenbankanfragen ergibt einen Compiler, der den Einsatz von relationalen Datenbanksystemen als effiziente Laufzeitumgebungen für nicht-relationale Sprachen ermöglicht.

# Abstract

Flat, unordered table data and a declarative query language established today's success of relational database systems. Provided with the freedom to choose the evaluation order and underlying algorithms, their complex query optimizers are geared to come up with the best execution plan for a given query. With over 30 years of development and research, relational database management systems belong to the most mature and efficient query processors (especially for substantial amounts of data).

In contrast, *ordered lists of possibly nested data structures* are used throughout in programming languages. Most developers handle these constructs on a daily basis and need to change their programming style, when confronted with a relational database system. To leverage the potential of the relational query processing facility in the context of non-relational languages—without the need for a context switch—we provide a query language that copes with order, nesting, and more complex data structures (such as tuples, named records, and XML data). Queries expressed in this language are compiled into relational queries over flat, unordered tables.

We take great care in the faithful mapping of the "alien" language constructs. This work describes the *Pathfinder* compiler achieving the transformation based on the *loop lifting* compilation strategy. The compiler transforms the input queries into logical algebra plans. The operators of this *unordered algebra* consist mainly of standard table algebra operators. Additional numbering and XML operators generate surrogate values to ensure an accurate representation of order, nesting, and more complex data structures.

Because of the exotic plan shape—the average plan consists of hundreds of operators and a large number of sharing points—and the numbering operators, the query optimizers of all tested database systems fail to come up with an efficient execution plan. We faced this challenge ourselves and describe an optimization framework that annotates the operators with plan properties and performs *local rewrites* based on the collected properties. The optimizations are guided by heuristics that are geared to significantly decrease the number of operators and order constraints.

The resulting plans are either turned into declarative SQL queries or scripts written for more specialized relational database systems, most notable MonetDB/XQuery. Whereas MonetDB/XQuery ships with a highly tuned runtime for XML processing, the generated SQL code allows to additionally benefit from the builtin optimizers of relational database systems. Experiments as well as plan analyses show that, in contrast to the initial plans, most optimized plans resemble queries from an ordinary database workload and are evaluated efficiently.

In summary, the compiler turns relational database systems into high-performance query processors for non-relational languages.

# Contents

**Zusammenfassung** **iii**

**Abstract** **v**

**1 Introduction** **1**

1.1 Optimizations in the Context of Pathfinder . . . 2
1.2 Logical Query Optimization . . . 5
1.3 Outline . . . 5
1.4 Prior Publications . . . 6

**2 Loop Lifted Query Compilation** **7**

2.1 Loop Lifting . . . 8
2.2 An Algebra for Non-Relational Languages . . . 9
2.3 Algebraic List Processing . . . 11
2.3.1 The Compilation Scheme . . . 12
2.3.2 Literals and Lists . . . 13
2.3.3 Iteration . . . 13
2.3.4 Operations and Conditionals . . . 14
2.3.5 Loop Lifting in Action . . . 15
2.4 More Order Constructs . . . 17
2.5 Adding Data Diversity . . . 18
2.5.1 Support for XML Data . . . 19
2.5.2 An Ordered View of Tables . . . 20
2.5.3 A Uniform Variant of SQL/XML . . . 22
2.6 A Flat Representation of Nesting . . . 22
2.6.1 (Un)Boxing . . . 25
2.6.2 Compilation of Nesting . . . 27
2.6.3 Nesting in Action . . . 29
2.7 Summary and Related Work . . . 29
2.7.1 Logical Algebra and Order . . . 31
2.7.2 Representing Nesting . . . 32
2.7.3 More Loop Lifting . . . 32

**3 Back-End Code Generation 35**
3.1 MonetDB/XQuery . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.1.1 MonetDB . . . . . . . . . . . . . . . . . . . . . . . . . 36
3.1.2 MonetDB Interpreter Language . . . . . . . . . . . . . . 37
3.1.3 Plan Generation for MonetDB . . . . . . . . . . . . . . . 37
3.1.4 MIL Code Generation . . . . . . . . . . . . . . . . . . . 40
3.2 SQL Code Generation . . . . . . . . . . . . . . . . . . . . . . . . 40
3.2.1 Translating Algebra Plans to SQL . . . . . . . . . . . . . 41
3.3 A First Quantitative Assessment . . . . . . . . . . . . . . . . . . 42
3.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**4 Logical Query Optimization 47**
4.1 Peephole Optimization . . . . . . . . . . . . . . . . . . . . . . . 48
4.2 House Cleaning . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.2.1 Empty Table Propagation . . . . . . . . . . . . . . . . . 50
4.2.2 Standard Operator Simplification . . . . . . . . . . . . . 51
4.2.3 An Alternative to Selection Pushdown . . . . . . . . . . . 58
4.2.4 An Alternative to Projection Pushdown . . . . . . . . . . 62
4.2.5 Common Subplan Elimination . . . . . . . . . . . . . . . 65
4.2.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . 70
4.3 Order Minimization . . . . . . . . . . . . . . . . . . . . . . . . . 71
4.3.1 Order Simplification . . . . . . . . . . . . . . . . . . . . 71
4.3.2 Numbering Operator Conversion . . . . . . . . . . . . . . 73
4.3.3 Order Pushup . . . . . . . . . . . . . . . . . . . . . . . . 74
4.3.4 Related Work . . . . . . . . . . . . . . . . . . . . . . . . 76
4.4 Query Unnesting . . . . . . . . . . . . . . . . . . . . . . . . . . 78
4.4.1 Equi-Join Pushdown . . . . . . . . . . . . . . . . . . . . 78
4.4.2 Distinct Operator Relocation . . . . . . . . . . . . . . . . 81
4.4.3 Dependency Disentanglement . . . . . . . . . . . . . . . 83
4.4.4 Related Work . . . . . . . . . . . . . . . . . . . . . . . . 86
4.5 Improving XML Queries . . . . . . . . . . . . . . . . . . . . . . . 87
4.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

**5 Optimization Assessment 91**
5.1 Analyzing the Query Plan Shape . . . . . . . . . . . . . . . . . . 91
5.1.1 Detection of Value-Based Joins . . . . . . . . . . . . . . . 92
5.1.2 Query Formulation . . . . . . . . . . . . . . . . . . . . . 96
5.2 Quantitative Evaluation . . . . . . . . . . . . . . . . . . . . . . . 99
5.2.1 XMark on MonetDB/XQuery . . . . . . . . . . . . . . . . 100
5.2.2 Join Graph Isolation on DB2 . . . . . . . . . . . . . . . . 102
5.2.3 Loop Lifting an Ordinary Database Workload . . . . . . . 107

**6 Summary and Outlook 111**
6.1 Contributions . . . . . 111
6.1.1 Optimization of Large Query Plans . . . . . 112
6.1.2 Order Minimization . . . . . 112
6.1.3 Unnesting of Loop Lifted Query Plans . . . . . 112
6.1.4 MonetDB/XQuery and Pathfinder . . . . . 113
6.2 Conclusion . . . . . 113
6.3 Open Problems and Future Work . . . . . 114
6.3.1 More XML Related Optimizations for XQuery . . . . . 114
6.3.2 Surrogate Values in Optimized Query Plans . . . . . 114
6.3.3 Automatization of Rewrites . . . . . 115

**Bibliography 117**

**Acknowledgments 127**

# Chapter 1

# Introduction

Relational database systems are among the best execution engines, when it comes to the processing of substantial amounts of data. Over the last 30 years, relational query processing infrastructure has been tuned to excel at the evaluation of regular-shaped table data. Most relational database systems rely on the complex query rewrite facilities of their optimizers to speed up the query evaluation. These optimizers have been designed to exploit the intricacies of the flat, unordered data model.

Vast amounts of software connect to relational database systems and issue queries against it. Although the language of choice to access a database system probably is the host language a program is implemented in, a developer typically needs to write database queries using the declarative language SQL. The developer needs to substitute the rich data model of the host language for SQL's flat, unordered data model and adjust himself to the new query language.

Although there exist several approaches that try to hide the database layer, their disadvantages are hard to ignore. The most prominent approach are *object-relational mappings*, which make database tables available as lists of host language objects. A developer may use the host language to query the data. Most generated SQL queries, however, retrieve only base table data and perform the remaining computation on the programming language's heap. The object-relational mappings thus ignore the advanced query facilities of a database system and misuse the database system as a data storage.

We are interested to supply a side effect free query language embedded in the host language that may query *ordered lists of possibly nested data structures* and at the same time employs a relational database system to evaluate the *complete* query. In the best case, we are able to evaluate the whole language on a database system: XQuery [16] is such a language. In all other cases, a language subset might suffice to conveniently express complex queries without leaving the host language. Microsoft's Linq [88] represents such a sensible language subset and is integrated into C# and Visual Basic. Linq already ships with a query provider that generates SQL queries. This Linq provider however fails to correctly cope with the ordered data model in a variety of cases [110].

Here, we describe how to narrow the gap between programming languages and relational database systems. We describe the internals of the *Pathfinder compiler*[1], which faithfully copes with ordered lists of possibly nested data structures. Pathfinder was originally designed to compile XQuery queries into both SQL queries or algebra queries targeting the main-memory database system MonetDB [18]. Pathfinder drives the XQuery to algebra compilation of the high-performance XQuery processor MonetDB/XQuery [21], but also performs the SQL code generation for a number of non-relational languages such as Haskell [98], Linq [88], and Ruby [105], which turn the back-end database system into an integral part of the program execution.

Pathfinder compiles an input language into an unordered logical table algebra based on the *loop lifted* compilation scheme [63], which ensures a faithful mapping of order, nesting, and data structures such as tuples, named records, and XML trees to tabular-shaped data. As the relational back-ends are overwhelmed by the size—an average query plan consists of hundreds of operators—and the unusual plan shape—the resulting query plans form directed acyclic graphs (DAGs)—a optimization phase becomes necessary before the back-end code is generated.

To ensure the acceptance of a relational database back-end as evaluation runtime, we need to guarantee its performance. In the scope of this work, we therefore only sketch the loop lifted compilation and the back-end code generation and focus on the optimization of the relational algebra plans.

## 1.1 Optimizations in the Context of Pathfinder

The *Pathfinder project* stimulated a fair amount of research topics. While the first topics in 2002 revolved around efficient XPath processing, the focus shifted in 2004 to the compilation of XQuery. Based on the loop lifted compilation, many sub-projects spawned touching compilation, optimization, and runtime topics (Figure 1.1). All of the subjects rely on the performance of the relational back-end and thus indirectly on the complexity of the resulting algebra plans.

In the following, we briefly sketch how the optimization of the logical algebra plans affect the various topics:

① **XPath Accelerator** [50, 62, 67]. The research articles about the XPath acceleration mainly discuss relational node encodings and the efficient evaluation of XPath steps. In [62], however, we relied on the optimization of existential quantifications to observe early-out semantics in the back-end's execution plans.

② **Staircase Join** [65, 66, 84]. The family of Staircase Join algorithms require

[1] `http://www.pathfinder-xquery.org`

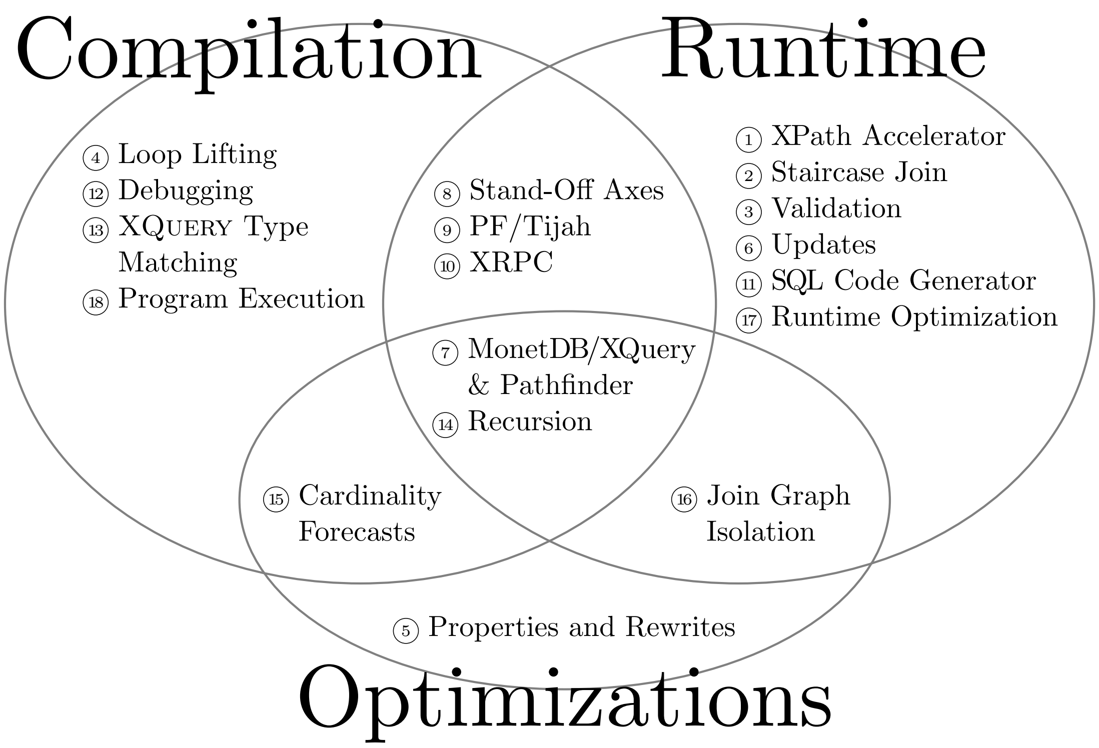


Figure 1.1: Research topics in the Pathfinder project categorized by the three main themes: Compilation, Optimizations, and Runtime. (Topics in overlapping regions touch multiple subjects.)

an ordered input and therefore may benefit from the removal of conflicting order constraints.

③ **Schema Validation** [52].

④ **Loop Lifting** [57,63,64]. The plan shape of the loop lifted algebra plans is best described as exotic: A vast number of operators and large number of sharing points can be observed. Relational database systems are not prepared to cope with such a workload and fail to deliver results in a reasonable time. Optimizations can significantly decrease the number of operators and sharing points.

⑤ **Properties and Rewrites** [51,61,103].

⑥ **XQuery Updates** [19,23]. XQuery's updates are embedded in normal queries and therefore benefit from efficient query evaluation.

⑦ **MonetDB/XQuery & Pathfinder** [20,21,117,120]. MonetDB/XQuery—the combination of the compiler Pathfinder and the relational back-end

MonetDB integrating runtime XML support such as the Staircase Join algorithms—requires the optimizations to avoid Cartesian products, which stem from the iterative evaluation of nested loops.

⑧ **Stand-Off Axes** [8–10].

⑨ **PF/Tijah** [68]. The text retrieval extension PF/Tijah relies on the performance of MonetDB/XQuery.

⑩ **XRPC** [126–128]. The distributed XQuery extension XRPC relies on the performance of MonetDB/XQuery.

⑪ **SQL Code Generator** [55]. The size of the generated SQL code directly corresponds to the size of the query plan. Some database systems struggle with the sheer size of the query text for unoptimized loop lifted algebra plans.

⑫ **Declarative Debugging** [58, 60]. A debugger relies on its interactivity. Therefore the performance of the relational back-end plays an important role.

⑬ **XQuery Type Matching** [118]. XQuery type matching extends the loop lifted compilation and thus introduces additional operators. As the type matching is employed only in some queries, optimizations may selectively remove this overhead.

⑭ **Recursion** [4,5]. We extended the loop lifted compilation with a fixed point recursion. Code for a more efficient version of this recursion is integrated, if the recursion body is *distributive*—a property that may be expressed by rewrite rules.

⑮ **Cardinality Forecasts** [119]. Cardinality misestimations increase with the number of plan operators. Optimizations that decrease the operator count therefore improve the cardinality forecasts.

⑯ **Join Graph Isolation** [53, 54].

⑰ **Runtime Optimization** [74, 75]. The runtime optimizations require *isolated join graphs*—bundles of relations interconnected with join predicates—as input. Such isolated join graphs are available only after optimizations have been performed.

⑱ **Program Execution** [56, 59, 110]. The compilation of ordered, nested data structures leads to plans that are very much alike to the plans in ④. Consequently, optimizations are mandatory to turn the relational back-end into an efficient query processor.

## 1.2 Logical Query Optimization

Finding the optimal query plan is a very challenging topic, as the potential search space is affected by both the number of operators and rewrites. Query optimization in database systems therefore ignores large parts of the search space: Query optimization is limited to finding *better* plans.

Whereas physical query optimization can compare two query plans based on their *cost* and choose the better one, there is no absolute comparison possible in logical query optimization. Minimizing the number of operators and the number of ordering constraints might be worthwhile, but does not guarantee a better query plan.

As SQL belongs to our back-end languages of choice, we are limited to logical query optimization and can follow only *heuristics* that lead to good query plans for many queries. A large share of the optimizations discussed in this work therefore consist of query plan simplifications, which, for example, decrease the number of operators and ordering constraints.

## 1.3 Outline

This work discusses the three main compilation stages of Pathfinder (a) loop lifted compilation, (b) optimization of the logical algebra plans, and (c) back-end code generation. In Chapter 2 we combine the ideas from the research topics ④ and ⑱ to describe the loop lifted compilation of ordered and possibly nested sequences of atomic values, tuples, and XML fragments into a logical table algebra. Chapter 3 discusses the back-end code generation for the relational XQuery back-end MonetDB/XQuery as well as the SQL code generation. A first experimental evaluation performed on MonetDB/XQuery and DB2 indicates the necessity of the optimizations for both back-ends.

The optimization of the logical algebra plans that stem from the loop lifted compilation is the topic of Chapter 4. To cope with query plans consisting of hundreds of operators, we apply a peephole-style rewrite strategy that performs local rewrites only. Three main heuristics, which perform (a) house cleaning, (b) order minimization, and (c) query unnesting, drive the optimizations. Whereas excerpts of the properties and rewrites were introduced in the publications of the topics ⑤ and ⑯, we substantially extend this set here and relate them to logical optimizations described in the database literature.

Various experiments in Chapter 5 demonstrate the effect of the optimizations. We assess the change of the operator distribution and the evaluation times of the rewritten queries. We however also analyze the effect of the rewrites from a different perspective and verify the robustness of the simplifications for syntactically differing, yet equivalent input queries. Finally, we turn the optimized query plans into SQL queries and observe the effectiveness of DB2's cost-based query optimizer, before Chapter 6 concludes this work.

## 1.4 Prior Publications

Results of this work are partially published in a number of scientific articles [20, 21, 53–59, 61, 103, 110]. Further publications indirectly benefit from the ideas and concepts described here [4, 5, 13, 19, 23, 60, 62]. I would like to thank Loredana Afanasiev, Stefan Aulbach, Peter Boncz, Simone Bonetti, Jan Flokstra, Torsten Grust, Dean Jacobs, Alfons Kemper, Stefan Manegold, Maarten Marx, Manuel Mayr, Sjoerd Mullender, Sherif Sakr, Tom Schreiber, Jens Teubner, and Maurice van Keulen for this fruitful collaboration.

# Chapter 2

# Loop Lifted Query Compilation

What does SQL and the *iteration primitives* in XQuery [16], Microsoft's Linq [88], Wadler's three-tier language Links [31], the purely functional language Haskell [98], the dynamic languages Ruby [105] and Python [100], and other languages [73] have in common? They share a common semantic ground: list comprehensions [123]. The iteration primitives of these *companion* languages describe the iterative evaluation of an expression $e_{body}$ under bindings of an unmodifiable *loop* or *iteration variable* (Figure 2.1).

| | | | |
|---|---|---|---|
| (XQuery) | `for $y in` $e_{in}$ `return` $e_{body}$ | (Ruby) | $e_{in}$`.collect{|y|` $e_{body}$`}` |
| (Linq) | $e_{in}$`.Select(y =>` $e_{body}$`)` | (Haskell) | `[` $e_{body}$ `| y <-` $e_{in}$ `]` |
| (Links) | `for (y <-` $e_{in}$`) [`$e_{body}$`]` | (Python) | `[` $e_{body}$ `for y in` $e_{in}$ `]` |

Figure 2.1: Iteration primitives in various companion languages.

Here, we are especially interested in loops without side-effecting computation such that the individual iterations may be evaluated *independently*. For SQL, XQuery, Links, and Haskell this is a given. For Linq, Ruby, and Python this requires programming discipline. As the individual iterations cannot interfere, the language processor may evaluate the loops in arbitrary order—or even in parallel.

The common semantic roots allow a closer interaction between database queries and iterative companion languages. We take advantage of an even deeper integration of database query functionality into programming languages (as exemplified by ActiveRecord or Ambition in the Ruby ecosystem [1, 11], Linq, and Links), in which selected iterative host programming language fragments may be translated into set-oriented algebraic programs. In what follows, we lay the groundwork for database-supported language runtimes that do not stumble if programs consume huge input data instances.

In the next section we shed light on a compilation technique, coined *loop lifting* in [63], which has been designed to let a relational database back-end directly participate in the evaluation of programs (or queries) written in an iterative style. The loop lifting compiler emits algebraic code for execution on the back-end, which then realizes the semantics of the input program. Loop lifting fully realizes the welcome independence of the iterated evaluations and enables the relational query engine to take advantage of its set-oriented processing paradigm.

We briefly discuss the relational operators and its semantics in Section 2.2, before we delve into the details of the compilation in Sections 2.3 to 2.6. We describe the compilation process based on a generic language LL[1] that combines the characteristics of the various companion languages (Section 2.3). In its basic variant LL operates only on ordered lists and thus provides a clear view on the loop lifting technique applied in the compilation. Subsequently, we extend LL as well as the compilation with sequence order modifications (Section 2.4), more complex data structures (Section 2.5), and nesting (Section 2.6).

## 2.1 Loop Lifting

The principal data structures in all companion languages except for SQL are *ordered* lists (sequences ($x_1$,$x_2$, ... ,$x_n$) of items $x_i$ in XQuery, values of type `[`$\alpha$`]`, `Array`, and `IEnumerable<T>` in Haskell, Ruby, and Linq, respectively). To properly reflect this order on the inherently unordered relational database back-end, we embed order in the data and use binary tables with columns pos|item (shown on the right) to represent such sequences. The values in column pos need not be dense and not even be of type `integer`; any ordered domain will do. In specific cases the required order may already reflected by the items $x_i$ themselves (think of a sequence of encoded nodes resulting from XPath location step evaluation in XQuery)—column item may then assume the role of pos.

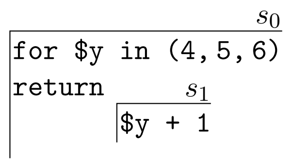
| pos | item |
|---|---|
| 1 | $x_1$ |
| 2 | $x_2$ |
| $\vdots$ | $\vdots$ |
| $n$ | $x_n$ |

The complete loop lifting approach [63] is centered around the iteration concept: An iteration primitive (Figure 2.1) introduces a new scope $s_i$, in which *any* subexpression is considered to be iteratively evaluated. The top level expression $e$ is assumed to be wrapped inside the scope $s_0$ of a pseudo single-iteration loop where the iteration variable does not occur free in $e$. The fundamental idea behind *loop lifting* is to produce algebraic code that consumes and emits a "fully unrolled" tabular representation of $e$'s value. Here, *unrolling* refers to the principle that

> a *single* ternary table with schema iter|pos|item holds the encoding of *all* values that $e$ assumes during its iterative evaluation.

Generally, such a table has key $\langle$iter, pos$\rangle$, since $e$ may yield a *sequence* of items in each distinct iteration. A row $\langle i, p, v\rangle$ in the table may invariantly be read as "in iteration $i$, expression $e$ yields item value $v$ at the sequence position corresponding

[1] In reference to the core concept of the compilation—loop lifting—we named the language LL.



```
                          s0
for $y in (4,5,6)
return      s1
         $y + 1
```

**(a) Query $Q_1$.**

(4, 5, 6) ($s_0$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 4 |
| 1 | 2 | 5 |
| 1 | 3 | 6 |

$y ($s_1$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 4 |
| 2 | 1 | 5 |
| 3 | 1 | 6 |

1 ($s_1$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 1 |
| 3 | 1 | 1 |

**(b) Associated intermediate result tables.**

Figure 2.2: Loop lifting in XQuery. Annotations $s_{0,1}$ denote iteration scopes.

to $p$'s rank in column pos". An empty list, in consequence, is represented by the absence of an iteration value $i$—in scope $s_0$ (with only a single iteration) an empty iter|pos|item table depicts the empty list.

Figure 2.2(a) depicts Query $Q_1$ together with its iteration scopes. The top-level expression `(4,5,6)` is evaluated in the pseudo scope $s_0$: all 3 sequence items have been produced in the first and only iteration of scope $s_0$ (iter $= 1$ in all rows), in the order indicated by column pos (leftmost table in Figure 2.2(b)). From this, the algebraic program derives that three iterations will be performed in the inner scope $s_1$: variable `$y` is bound to one integer in the sequence `(4,5,6)` in each iteration. The constant `1` is loop lifted and invariantly evaluates to `1` for all iterations (rightmost table in Figure 2.2(b)).

## 2.2 An Algebra for Non-Relational Languages

The compilation emits plans over a table algebra, whose operators (Table 2.1) have been selected to reflect the capabilities and execution model of modern SQL-based relational database management systems. Most of these operators are purist RISC-like style variants of classical algebra operators. The selection operator $\sigma_b$, for example, does not evaluate predicates on its own, but relies on the presence of a Boolean column $b$. Duplicates are removed explicitly by operator $\delta$.

The few non-textbook operators include the column attachment operator $@$, the family of numbering operators ($\overrightarrow{\#}$, $\overrightarrow{\Cup}$, and $\Cup$), and the placeholders for XML support. The former $@_{\mathsf{a}:val}(e)$ is equivalent to $(e) \times \begin{array}{|c|}\hline a \\ \hline val \\ \hline \end{array}$ and eases the optimization of constant columns.

The numbering operators enable the compilation to encode iteration, order, and nesting information in the data. The row numbering operator $\overrightarrow{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}$ is equivalent to the `ROW_NUMBER` clause in SQL:1999 [89]. For every group in $b_{grp}$, the row numbering operator provides a consecutive numbering in column $a$ starting from value 1, based on the ordering criteria $b_1,\ldots,b_n$. If no absolute order is present, an arbitrary order is chosen. Figures 2.3(a) and 2.3(b) exemplify the semantics of operator $\overrightarrow{\#}$.

Similarly the row ranking operator $\overrightarrow{\Cup}$ follows the semantics of the SQL:1999 `DENSE_RANK` clause [89]. In contrast to the row numbering, operator $\overrightarrow{\Cup}$ provides

| **Operator** | **Description** |
|---|---|
| $\pi_{a_1:b_1,\ldots,a_n:b_n}$ | projection and column renaming |
| $\sigma_b$ | select rows with $b = \mathsf{true}$ |
| $\times$ | Cartesion product |
| $\bar{\bowtie}_{b_1=b_2}$ , $\bar{\bowtie}_b$ | equi-join |
| $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}$ | theta-join ($\theta \in \{=,<,>,\leq,\geq,\neq\}$) |
| $\delta$ | duplicate elimination |
| $\dot{\cup}_{b_1,\ldots,b_n 2}$ , $\setminus_{b_1,\ldots,b_n 2}$ | disjoint union, difference (on columns $b_1,\ldots,b_n$) |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}$ | aggregation/grouping ($\circ \in \{\textsc{max}, \textsc{sum}, \textsc{count}, \ldots\}$) |
| $\begin{array}{\|c\|}\hline a_1 \ldots a_n \\ \hline \varnothing \\ \hline \end{array}$, $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|c\|}{a_1 \ldots a_n} \\ \hline & & \\ \hline \end{array}$ | (empty) literal table |
| $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|c\|}{name} \\ \hline a_1 & \ldots & a_n \\ \hline \end{array}$ | database table reference |
| $@_{a:val}$ | column attachment (with constant value *val*) |
| $\circledcirc_{a:\langle b_1,\ldots,b_n\rangle}$ | $n$-ary comparison/arithmetic/Boolean operator $\circ$ |
| $\mathrm{CAST}^{type}_{a:\langle b\rangle}$ | type casting |
| $\overrightarrow{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}$ | row numbering (with order $\langle b_1,\ldots,b_n\rangle$, grouping $b_{grp}$) |
| $\overrightarrow{\varrho}_{a:\langle b_1,\ldots,b_n\rangle}$ | row ranking (with order $\langle b_1,\ldots,b_n\rangle$) |
| $\varrho_{a:\langle b_1,\ldots,b_n\rangle}$ | (arbitrary) row ranking (with order $\langle b_1,\ldots,b_n\rangle$) |
| $\lrcorner^{\alpha,v}_{a:\langle b\rangle}$ | XML path step-join (along axis $\alpha$ and node test $v$) |
| $\textsc{doc}_{a:\langle b\rangle}$ | XML document lookup |
| $\text{⚛}_{a:\langle b\rangle}$ | XML node atomization |
| $\text{♀}_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}$ | plan root (with payload columns $\langle b_3,\ldots,b_n\rangle$) |

Table 2.1: Table algebra dialect consisting of RISC-like relational operators. ($a$ represents fresh and $b$ existing input column names.)

consecutive numbers for every distinct group of values—equal ordering criteria lead to the same value (Figure 2.3(c)). In the loop lifted compilation scheme these two operators produce iteration identifiers, absolute position values as well as grouping identifiers.

The ranking operator $\varrho$, on the other hand, may generate relative positions and thus provides more optimization potential. $\varrho$ represents a less restrictive variant of $\overrightarrow{\varrho}$ that may construct values of any domain as long as they reflect the ranking criteria (Figure 2.3(d)).

The algebra supports XML processing independent of the underlying XML encoding. The algebra assumes the existence of *node surrogates* that implement the XML concepts of *node identity* and *document order*—for example, ORDPATH labels [95] or a variant of the preorder encodings [67, 80] serve this purpose. The XML tree knowledge is encapsulated in the three placeholder operators: DOC,

| res | $i_1$ | $i_2$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 1 | 3 | 3 |
| 2 | 3 | 3 |

**(a)** $\vec{\#}_{\mathsf{res}:\langle \mathsf{i}_2\rangle/\mathsf{i}_1}$.

| res | $i_1$ | $i_2$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 4 | 3 | 3 |
| 3 | 3 | 3 |

**(b)** $\vec{\#}_{\mathsf{res}:\langle \mathsf{i}_1,\mathsf{i}_2\rangle}$.

| res | $i_1$ | $i_2$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 3 |
| 3 | 3 | 3 |

**(c)** $\vec{\veebar}_{\mathsf{res}:\langle \mathsf{i}_1,\mathsf{i}_2\rangle}$.

| res | $i_1$ | $i_2$ |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 1 | 2 |
| 4 | 3 | 3 |
| 4 | 3 | 3 |

**(d)** $\veebar_{\mathsf{res}:\langle \mathsf{i}_1,\mathsf{i}_2\rangle}$.

Figure 2.3: Semantics of the numbering operators. (Column res depicts the differences.)

⏌, and ⚛. For every input string, the document operator DOC retrieves the corresponding document node surrogate $\gamma$. For every node surrogate $\gamma$, the atomization operator ⚛ returns a single string that stores the (concatenated) text content of the corresponding XML subtree.

XML tree traversal—in the style of XPath—is performed by ⏌. The XML path step join operator $⏌^{\alpha,v}_{a:\langle b\rangle}$ consumes column $b$, which contains the node surrogates $\gamma_i$, and applies a structural join with a *universal* XML document table. The structural join predicate is described by the axis $\alpha$ and the node test $v$. The *flat join result* features all input columns and a new result column $a$, which stores the matching node surrogates. In contrast to the XPath step semantics, ⏌ does not remove duplicates and guarantees no output order.[2]

The serialization operator $♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}$ forms the plan root of every query plan. ♀ stores the roles of its input columns: column $b_1$ represents the grouping information, column $b_2$ the sequence order, and columns $\langle b_3,\ldots,b_n\rangle$ store the payload information.

## 2.3 Algebraic List Processing

While the companion languages XQuery, Linq, Links, Haskell, Ruby and Python share their side effect free iteration semantics, they differ in their syntactic form (Figure 2.1) and their data types. XQuery, for example, operates on flat item sequences, whereas the other languages handle records and nesting. We deploy the loop lifting compilation scheme of the generic language LL, which represents the common aspects of all companion languages. Language extensions (described in the following sections) then account for the varying aspects of the companion languages.

Figure 2.4 describes a basic variant of LL that supports atomic values, arithmetics, comparisons, (empty) lists, iteration, conditionals, and aggregates. In this basic variant LL incorporates XQuery's sequence semantics: All lists are flat—for example, the *ForExpr* returns a flat list—and a constant value $v$ is equivalent to

[2]Peek forward to Query $Q_3$ at page 22 for an example involving XML operators.

$$
\begin{array}{lcl}
\mathit{Expr} & ::= & \mathit{Integer} \mid \mathit{Boolean} \mid \mathit{String} \\
 & \mid & \texttt{(}\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}\texttt{)} \\
 & \mid & \texttt{empty} \\
 & \mid & \texttt{append (}\mathit{Expr}\ (\texttt{,}\ \mathit{Expr})^{*}\texttt{)} \\
 & \mid & \mathit{ForExpr} \\
 & \mid & \mathit{Var} \\
 & \mid & \mathit{IfExpr} \\
 & \mid & \mathit{AggrExpr} \\
\mathit{ForExpr} & ::= & \texttt{for}\ \mathit{Var}\ \texttt{in}\ \mathit{Expr}\ \texttt{return}\ \mathit{Expr} \\
\mathit{Var} & ::= & \texttt{\$}\mathit{Identifier} \\
\mathit{IfExpr} & ::= & \texttt{if (}\mathit{Expr}\texttt{) then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} \\
\mathit{AggrExpr} & ::= & \texttt{count (}\mathit{Expr}\texttt{)} \\
\mathit{Integer} & ::= & \texttt{-}^{?}[\texttt{0-9}]^{+} \\
\mathit{Boolean} & ::= & \texttt{true} \mid \texttt{false} \\
\mathit{String} & ::= & \texttt{"}\ldots\texttt{"} \\
\mathit{Op} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{div} \mid \texttt{=} \mid \texttt{>} \mid \texttt{<} \mid \ldots \\
\mathit{Identifier} & ::= & [\texttt{A-Za-z0-9}]^{+}
\end{array}
$$

Figure 2.4: Grammar of the input language (basic LL).

a list containing the single constant value: `[`$v$`]`[3].

### 2.3.1 The Compilation Scheme

The compilation process from LL to the table algebra of Section 2.2 is described in terms of *inference rules*. An inference rule of the form

$$\Gamma; \mathit{loop} \vdash e \Mapsto q$$

implements a mapping where, given a variable environment $\Gamma$ and the iteration context *loop*, a LL expression $e$ compiles to an algebraic query plan $q$. $\Gamma$ holds the algebraic query plans that represent the free variables that are currently in scope. The iteration context *loop* refers to a query plan representing all available iterations (in column iter).

The compilation process starts with an empty environment $\Gamma$, the singleton *loop* table $\begin{array}{|c|}\hline \text{iter} \\ \hline 1 \\ \hline \end{array}$ (encoding the iteration scope $s_0$ discussed in Section 2.1), and the top-level LL expression. The inference rules are truly compositional, as any rule invariantly expects and provides the same table format—a table with an $\langle \text{iter}, \text{pos}, \text{item} \rangle$ schema.

[3]We use square brackets (`[...]`) to depict list-valued results.

### 2.3.2 Literals and Lists

The algebraic representation of literal values depends on the iteration context *loop*: A constant value is “loop lifted” to all iterations. For every iteration Rule Const produces a singleton sequence containing the literal *val*.

$$\frac{}{\Gamma; \mathit{loop} \vdash \mathit{val} \Rrightarrow @_{\mathsf{pos}:1}\left(@_{\mathsf{item}:\mathit{val}}\left(\mathit{loop}\right)\right)}\,(\textsc{Const})$$

Likewise, a literal empty sequence is mapped to all iterations. Its evaluation however will always result in an empty table.

$$\frac{}{\Gamma; \mathit{loop} \vdash \texttt{empty} \Rrightarrow \mathit{loop} \times \begin{array}{|cc|}\hline \mathsf{pos} & \mathsf{item} \\ \hline \multicolumn{2}{|c|}{\varnothing} \\ \hline \end{array}}\,(\textsc{Empty})$$

In LL, a list is constructed with append. Rule Append dispatches the compilation for the $n$ arguments, before it concatenates the resulting $n$ query plans with union operators. While the correct order within the arguments is encoded in column pos, column ord stores the concatenation order of the $n$ sequences. Together columns ord and pos correctly describe the overall sequence order. To guarantee the compositionality—an $\langle \mathsf{iter}, \mathsf{pos}, \mathsf{item} \rangle$ output schema—the two columns are merged into a new position column $\mathsf{pos}_{new}$ by operator $\curlyvee$.

$$\frac{\begin{array}{l} \Gamma; \mathit{loop} \vdash e_1 \Rrightarrow q_1 \\ q_{1\cdot ext} \equiv @_{\mathsf{ord}:1}(q_1) \\ {\scriptstyle i=2,\ldots,n}\left| \begin{array}{l} \Gamma; \mathit{loop} \vdash e_i \Rrightarrow q_i \\ q_{i\cdot ext} \equiv q_{(i-1)\cdot ext} \,\dot{\cup}_{\mathsf{iter},\mathsf{ord},\mathsf{pos},\mathsf{item}} \left(@_{\mathsf{ord}:i}(q_i)\right) \end{array}\right. \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}_{new},\mathsf{item}}\left(\curlyvee_{\mathsf{pos}_{new}:\langle \mathsf{ord},\mathsf{pos}\rangle}(q_{n\cdot ext})\right) \end{array}}{\Gamma; \mathit{loop} \vdash \texttt{append (}e_1\texttt{, ... ,}e_n\texttt{)} \Rrightarrow q}\,(\textsc{Append})$$

### 2.3.3 Iteration

The `for`-loop is the centerpiece of the loop lifting compiler. Based on the algebraic representation of the binding expression $e_{in}$, Rule For generates fresh iteration values (operator $\vec{\#}$). This numbering becomes the new iteration context $\mathit{loop}_v$ on which all free variables during the compilation of the return part $e_{return}$ depend. In addition to the already existing free variables, which are loop lifted to the new iteration scope, the fresh environment $\Gamma_v$ stores the mapping from `$v` to its algebraic representation $q_v$.

After the compilation of the body expression $e_{return}$, an equi-join aligns the former iteration values in column outer with the resulting plan. The replacement of the inner iter column by the previous iteration column outer re-establishes the former iteration scope. Furthermore, similar to Rule Append, the values of the position column pos are re-calculated to correctly reflect the sequence order.

$$\frac{\begin{array}{l}\{\ldots, x \mapsto q_x, \ldots\}\,;\, \mathit{loop} \vdash e_{\mathit{in}} \Rrightarrow q_{\mathit{in}} \\ q_{\mathit{in}\cdot \mathit{ext}} \equiv \vec{\#}_{\mathsf{inner}:\langle\mathsf{iter},\mathsf{pos}\rangle}(q_{\mathit{in}}) \\ q_v \equiv @_{\mathsf{pos}:1}\,(\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{item}}(q_{\mathit{in}\cdot \mathit{ext}})) \\ \mathit{loop}_v \equiv \pi_{\mathsf{iter}:\mathsf{inner}}(q_{\mathit{in}\cdot \mathit{ext}}) \\ \mathit{map}_v \equiv \pi_{\mathsf{outer}:\mathsf{iter},\mathsf{inner},\mathsf{pos}_{\mathit{out}}:\mathsf{pos}}(q_{\mathit{in}\cdot \mathit{ext}}) \\ \Gamma_v \equiv \{\ldots, x \mapsto \pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},\mathsf{item}}(q_x \overset{=}{\bowtie}_{\mathsf{iter}=\mathsf{outer}} \mathit{map}_v), \ldots\} \\ \qquad + \{\mathtt{\$v} \mapsto q_v\} \\ \Gamma_v; \mathit{loop}_v \vdash e_{\mathit{return}} \Rrightarrow q_{\mathit{return}} \\ q_{\mathit{return}\cdot \mathit{ext}} \equiv q_{\mathit{return}} \overset{=}{\bowtie}_{\mathsf{iter}=\mathsf{inner}} \mathit{map}_v \\ q \equiv \pi_{\mathsf{iter}:\mathsf{outer},\mathsf{pos}:\mathsf{pos}_{\mathit{new}},\mathsf{item}}\,(\varrho_{\mathsf{pos}_{\mathit{new}}:\langle\mathsf{pos}_{\mathit{out}},\mathsf{pos}\rangle}\,(q_{\mathit{return}\cdot \mathit{ext}}))\end{array}}{\{\ldots, x \mapsto q_x, \ldots\}\,;\, \mathit{loop} \vdash \mathtt{for\ \$v\ in}\ e_{\mathit{in}}\ \mathtt{return}\ e_{\mathit{return}} \Rrightarrow q}\,(\textsc{For})$$

As Rule For maintains the correct representation of all free variables, a variable lookup simply results in a reference to the corresponding algebraic query plan.

$$\frac{}{\{\ldots, \mathtt{\$v} \mapsto q, \ldots\}\,;\, \mathit{loop} \vdash \mathtt{\$v} \Rrightarrow q}\,(\textsc{Var})$$

### 2.3.4 Operations and Conditionals

For binary operations such as arithmetics or comparisons, an equi-join positionally aligns the rows of the two input algebra plans on column iter (Rule Op). Subsequently, operation $\circ$ is performed row-wise by the corresponding operator $\circledcirc$.

$$\frac{\begin{array}{l}\Gamma; \mathit{loop} \vdash e_1 \Rrightarrow q_1 \\ \Gamma; \mathit{loop} \vdash e_2 \Rrightarrow q_2 \\ q \equiv \circledcirc_{\mathsf{res}:\langle\mathsf{item},\mathsf{item}_r\rangle}\,(q1 \overset{=}{\bowtie}_{\mathsf{iter}}\,(\pi_{\mathsf{iter},\mathsf{item}_r:\mathsf{item}}(q_2)))\end{array}}{\Gamma; \mathit{loop} \vdash \mathtt{(}e_1 \circ e_2\mathtt{)} \Rrightarrow \pi_{\mathsf{iter},\mathsf{pos},\mathsf{item}:\mathsf{res}}(q)}\,(\textsc{Op})$$

Aggregates, which consume a list of values, map to relational aggregates (Rule Count). The correct treatment of empty lists, however, requires special care. Based on the iteration context *loop*, empty sequences are detected ($q_{\mathit{empty}}$) and their replacement is added to the remaining aggregate results ($q_{\mathit{full}}$).

$$\frac{\begin{array}{l}\Gamma; \mathit{loop} \vdash e \Rrightarrow q \\ q_{\mathit{aggr}} \equiv \mathrm{GRP}_{\mathsf{item}:\textsc{count}(*)/\mathsf{iter}}(q) \\ q_{\mathit{empty}} \equiv @_{\mathsf{item}:0}\,(\mathit{loop} \setminus_{\mathsf{iter}}\,(q_{\mathit{aggr}})) \\ q_{\mathit{full}} \equiv @_{\mathsf{pos}:1}\,(q_{\mathit{aggr}} \dot{\cup}_{\mathsf{iter},\mathsf{item}}\, q_{\mathit{empty}})\end{array}}{\Gamma; \mathit{loop} \vdash \mathtt{count\ (}e\mathtt{)} \Rrightarrow q_{\mathit{full}}}\,(\textsc{Count})$$

Rule IfThenElse compiles conditional expressions. Query plan $q_{\mathit{if}}$ is used to split the iteration context based on the values in column item into two groups. Because the new iteration contexts $\mathit{loop}_{\mathit{then}}$ and $\mathit{loop}_{\mathit{else}}$ are disjoint, the query plans for the two branches $q_{\mathit{then}}$ and $q_{\mathit{else}}$ both operate on their restricted, distinct iteration subset.

$$\frac{\begin{array}{l}\{\ldots, x \mapsto q_x, \ldots\}\,;\mathit{loop} \vdash e_{\mathit{if}} \Rrightarrow q_{\mathit{if}} \\ \mathit{loop}_{\mathit{then}} \equiv \pi_{\mathsf{iter}}(\sigma_{\mathsf{item}}(q_{\mathit{if}})) \\ \Gamma_{\mathit{then}} \equiv \{\ldots, x \mapsto q_x \mathbin{\bar{\ltimes}_{\mathsf{iter}}} \mathit{loop}_{\mathit{then}}, \ldots\} \\ \Gamma_{\mathit{then}}; \mathit{loop}_{\mathit{then}} \vdash e_{\mathit{then}} \Rrightarrow q_{\mathit{then}} \\ \mathit{loop}_{\mathit{else}} \equiv \pi_{\mathsf{iter}}(\sigma_{\mathsf{item}_{\mathit{neg}}}(\ominus_{\mathsf{item}_{\mathit{neg}}:\langle\mathsf{item}\rangle}(q_{\mathit{if}}))) \\ \Gamma_{\mathit{else}} \equiv \{\ldots, x \mapsto q_x \mathbin{\bar{\ltimes}_{\mathsf{iter}}} \mathit{loop}_{\mathit{else}}, \ldots\} \\ \Gamma_{\mathit{else}}; \mathit{loop}_{\mathit{else}} \vdash e_{\mathit{else}} \Rrightarrow q_{\mathit{else}} \end{array}}{\{\ldots, x \mapsto q_x, \ldots\}\,;\mathit{loop} \vdash \begin{array}{l}\texttt{if (}e_{\mathit{if}}\texttt{)} \\ \texttt{then } e_{\mathit{then}} \\ \texttt{else } e_{\mathit{else}}\end{array} \Rrightarrow q_{\mathit{then}} \mathbin{\dot{\cup}_{\mathsf{iter,pos,item}}} q_{\mathit{else}}} \;(\textsc{IfThenElse})$$

## 2.3.5 Loop Lifting in Action

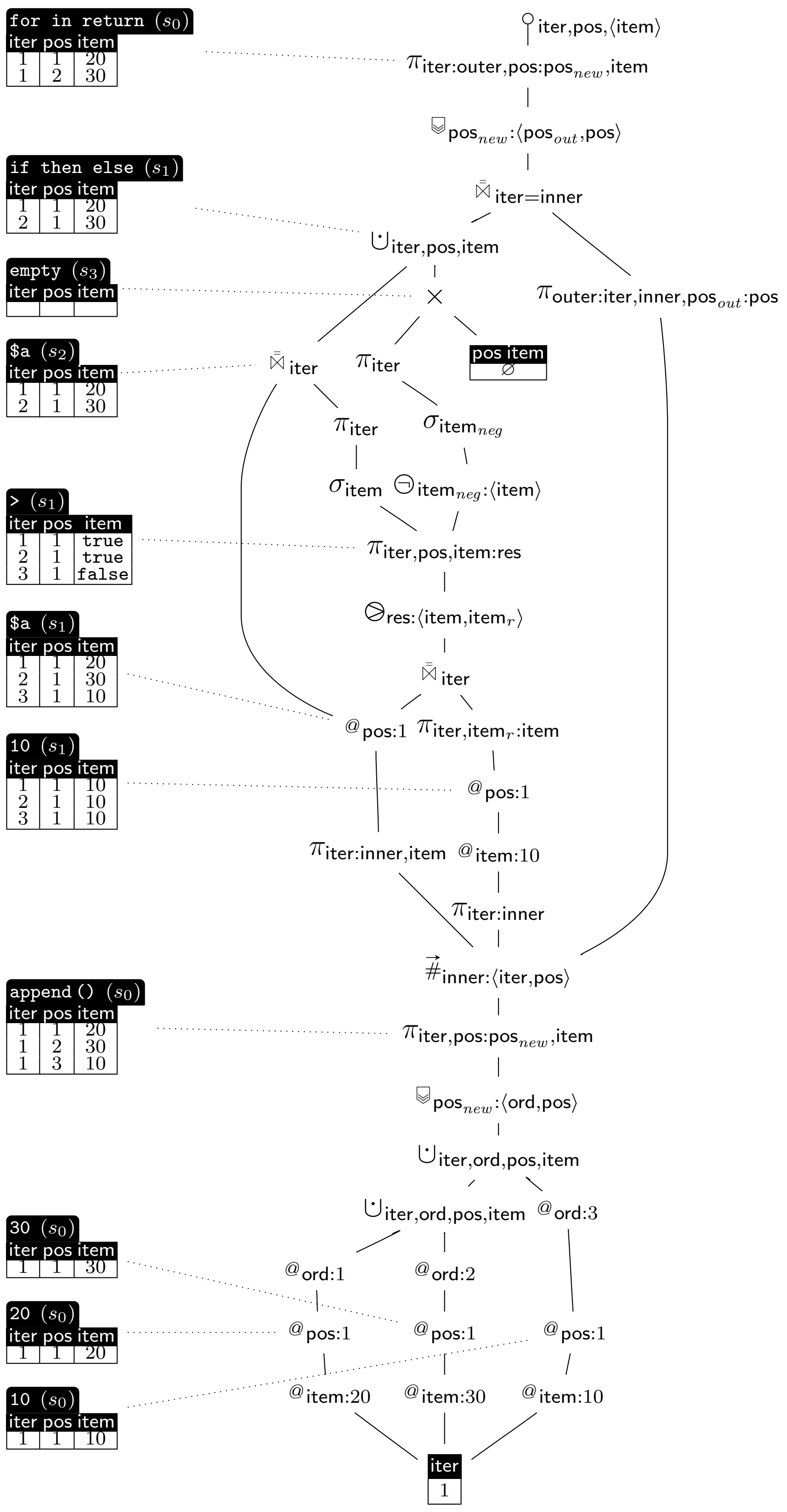
With the basic variant of LL we can already build atomic lists, perform comparisons and arithmetics, and apply filters to them. One query instance that takes these concepts into account is Query $Q_2$. The query builds a literal list `[20, 30, 10]`, filters out the third iteration binding (`10`), and returns the flat literal sequence `[20, 30]`.

```
                                                  s0
for $a in append(20,30,10)                       s1
return   if (($a > 10)) then $a else empty
                              s2        s3
```
$(Q_2)$

Query $Q_2$ is annotated with iteration scopes ($s_0, \ldots, s_3$). These scopes mark the places where the iteration context *loop* changes. In scope $s_0$, for example, only a single iteration is active, whereas in scope $s_2$ there exist two iterations—namely for the bindings `$a` = `20` and `$a` = `30`.

Figure 2.5 shows the algebraic query plan generated for Query $Q_2$. For every inference rule application we attached an annotation that shows the query construct, the scope it is evaluated in, and the intermediate result that would be observed.

The effect of loop lifting can be observed by comparing the two different representations of the constant value `10`: In nested scopes (here scope $s_1$), expressions are compiled in dependence of the current iteration context in a fully unrolled fashion.



for in return ($s_0$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 20 |
| 1 | 2 | 30 |

if then else ($s_1$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 20 |
| 2 | 1 | 30 |

empty ($s_3$)

| iter | pos | item |
|---|---|---|
| | | |

$a ($s_2$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 20 |
| 2 | 1 | 30 |

> ($s_1$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | true |
| 2 | 1 | true |
| 3 | 1 | false |

$a ($s_1$)

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

10 ($s_1$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 10 |
| 2 | 1 | 10 |
| 3 | 1 | 10 |

append () ($s_0$)

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

30 ($s_0$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 30 |

20 ($s_0$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 20 |

10 ($s_0$)

| iter | pos | item |
|---|---|---|
| 1 | 1 | 10 |

Figure 2.5: Algebraic query plan encoding Query $Q_2$ with annotations. Tables on the left-hand side depict the corresponding query constructs, iteration scopes ($s_0, \ldots, s_3$), and intermediate result representations.

$$
\begin{array}{lll}
\textit{Expr} & ::= & \ldots \\
 & | & \texttt{reverse (}\textit{Expr}\texttt{)} \\
 & | & \texttt{unordered (}\textit{Expr}\texttt{)} \\
\ldots & & \\
\textit{ForExpr} & ::= & \ldots \\
 & | & \texttt{for } \textit{Var} \texttt{ at } \textit{Var} \texttt{ in } \textit{Expr} \\
 & & \texttt{order by } \textit{Expr} \ (\texttt{,}\ \textit{Expr})^{*} \texttt{ return } \textit{Expr} \\
\ldots & &
\end{array}
$$

Figure 2.6: Order constructs that extend the grammar of basic LL (Figure 2.4).

## 2.4 More Order Constructs

The first extension to the basic variant of LL adds list order manipulation. The extension provides means to change the sequence order, ignore it, and filter values based on their sequence position. Figure 2.6 extends the grammar of Figure 2.4 with the order-manipulating language constructs. Grammar rule *Expr* is extended with the two functions `reverse` and `unordered`. Grammar rule *ForExpr* has a second iteration primitive that provides position information and result ordering by means of a position variable (`at` *Var*) and an `order by` clause.

The inference rules for `reverse` and `unordered` manipulate the position column pos, but ignore the iteration and value columns. Rule Reverse uses a rank operator ⛉ with a descending ordering criterion to reverse the direction of the values in column pos. Rule Unordered generates a new pos column in an undefined order. The projection, discarding the previous pos column, is crucial as it serves as an indicator for subsequent optimizations: any order maintenance in the upstream plan may be removed.

$$\frac{\Gamma; \textit{loop} \vdash e \Mapsto q}{\Gamma; \textit{loop} \vdash \texttt{reverse (}e\texttt{)} \Mapsto \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}_{new},\mathsf{item}}\left(\text{⛉}_{\mathsf{pos}_{new}:\langle \mathsf{pos}:desc\rangle}(q)\right)}(\textsc{Reverse})$$

$$\frac{\Gamma; \textit{loop} \vdash e \Mapsto q}{\Gamma; \textit{loop} \vdash \texttt{unordered (}e\texttt{)} \Mapsto \vec{\#}_{\mathsf{pos}:\langle\rangle}\left(\pi_{\mathsf{iter},\mathsf{item}}(q)\right)}(\textsc{Unordered})$$

Rule OrderedFor extends Rule For and considers the position variable `$p` and the `order by` clause. For each iteration, the additional operator $\vec{\#}_{\mathsf{pos}_{abs}:\langle\mathsf{pos}\rangle/\mathsf{iter}}$ creates the absolute position values from the relative positions in column pos. The values created by this row numbering operator implement the positions bound to the position variable `$p`. Adding the mapping `$p` $\mapsto q_p$ to the variable environment makes the algebraic representation of the position variable visible during the compilation of `order by` and `return` clauses.

For each iteration, the compilation of the `order by` expressions $e_1, \ldots, e_n$ provides at most one order value. Missing values are handled similarly to empty lists in aggregates—compare $q_{i\cdot empty}$ with $q_{empty}$ in Rule Count on page 14. The plans $q_{i\cdot full}$ extend the mapping $map_v$ with additional order information. Ultimately, this order information is used to influence the output list order in operator $\curlyvee$.

Although the ordering criteria affect the order only inside the `for` loop (column $\mathsf{pos}_{out}$) they overrule the default order in column $\mathsf{pos}$. The $\mathsf{ord}_i$ columns ensure that the position values for missing values are ignored and empty lists are treated as if they were smaller than any other value.[4]

$$\dfrac{\begin{array}{l}
\{\ldots, x \mapsto q_x, \ldots\}\,;loop \vdash e_{in} \Mapsto q_{in} \\
q_{in\cdot ext} \equiv \vec{\#}_{\mathsf{inner}:\langle\mathsf{iter},\mathsf{pos}\rangle}\left(\vec{\#}_{\mathsf{pos}_{abs}:\langle\mathsf{pos}\rangle/\mathsf{iter}}(q_{in})\right) \\
q_v \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{item}}(q_{in\cdot ext})\right) \\
q_p \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{item}:\mathsf{pos}_{abs}}(q_{in\cdot ext})\right) \\
loop_v \equiv \pi_{\mathsf{iter}:\mathsf{inner}}(q_{in\cdot ext}) \\
map_v \equiv \pi_{\mathsf{outer}:\mathsf{iter},\mathsf{inner},\mathsf{pos}_{out}:\mathsf{pos}}(q_{in\cdot ext}) \\
\Gamma_v \equiv \{\ldots, x \mapsto \pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},\mathsf{item}}(q_x \overset{=}{\bowtie}_{\mathsf{iter}=\mathsf{outer}} map_v), \ldots\} \\
\qquad + \{\texttt{\$v} \mapsto q_v\} + \{\texttt{\$p} \mapsto q_p\} \\
map_0 \equiv map_v \\
{\scriptstyle i=1,\ldots,n}\left|\begin{array}{l}
\Gamma_v; loop_v \vdash e_i \Mapsto q_i \\
q_{i\cdot empty} \equiv @_{\mathsf{pos}:1}\left(@_{\mathsf{item}:1}\left(loop_v \setminus_{\mathsf{iter}} (q_i)\right)\right) \\
q_{i\cdot full} \equiv \left(@_{\mathsf{ord}:1}(q_{i\cdot empty})\right) \dot\cup_{\mathsf{iter},\mathsf{ord},\mathsf{pos},\mathsf{item}} \left(@_{\mathsf{ord}:2}(q_i)\right) \\
map_i \equiv map_{i-1} \overset{=}{\bowtie}_{\mathsf{inner}} \left(\pi_{\mathsf{inner}:\mathsf{iter},\mathsf{ord}_i:\mathsf{ord},\mathsf{pos}_i:\mathsf{item}}(q_{i\cdot full})\right)
\end{array}\right. \\
\Gamma_v; loop_v \vdash e_{return} \Mapsto q_{return} \\
q_{return\cdot ext} \equiv q_{return} \overset{=}{\bowtie}_{\mathsf{iter}=\mathsf{inner}} map_n \\
q_{return\cdot pos} \equiv \curlyvee_{\mathsf{pos}_{new}:\langle\mathsf{pos}_{out},\mathsf{ord}_1,\mathsf{pos}_1,\ldots,\mathsf{ord}_n,\mathsf{pos}_n,\mathsf{pos}\rangle}\left(q_{return\cdot ext}\right) \\
q \equiv \pi_{\mathsf{iter}:\mathsf{outer},\mathsf{pos}:\mathsf{pos}_{new},\mathsf{item}}\left(q_{return\cdot pos}\right)
\end{array}}{\{\ldots, x \mapsto q_x, \ldots\}\,;loop \vdash \begin{array}{l}\texttt{for \$v at \$p in } e_{in} \\ \texttt{order by } e_1\texttt{, ... ,}e_n \\ \texttt{return } e_{return}\end{array} \Mapsto q}\;(\text{OrderedFor})$$

## 2.5 Adding Data Diversity

Up til now all LL queries operated on lists of atomic values. Here, we extend LL with additional data types available in the various companion languages. XML data processing is one of the most important aspects of XQuery. An ordered view on database-resident tables by means of list of hashes, tuples, and named records is available in Ruby, Links, and Linq.

[4]This corresponds to XQuery's order option `empty least`.

$$
\begin{array}{lll}
\mathit{Expr} & ::= & \ldots \\
 & | & \texttt{doc}\,(\mathit{Expr}) \\
 & | & \mathit{Expr}\texttt{/}\alpha\texttt{::}n \\
 & | & \texttt{d-d-o}\,(\mathit{Expr}) \\
 & | & \texttt{value}\,(\mathit{Expr}) \\
\alpha & ::= & \texttt{child} \,|\, \texttt{descendant} \,|\, \texttt{parent} \,|\, \ldots \\
n & ::= & \texttt{element}\,(\mathit{ID}) \,|\, \texttt{text}\,() \,|\, \ldots \\
\ldots & &
\end{array}
$$

Figure 2.7: XML constructs extending the grammar of basic LL (Figure 2.4).

We first add support for XML data processing, before we extend the compilation scheme to cope with data structures consisting of multiple elements such as tuples.

## 2.5.1 Support for XML Data

The companion languages XQuery and Linq provide support for XML data processing. In both languages XML documents can be accessed, their tree structure can be traversed by XML path steps, and the text representation of XML nodes can be extracted. Figure 2.7 shows the XML constructs extending the LL grammar of Figure 2.4. `doc` provides the document lookup, $\texttt{/}\alpha\texttt{::}n$ and `d-d-o` represent the path step semantics, and `value` transforms the XML node content into a string.

As the path step semantics of XQuery and Linq differ slightly—an XPath location step consumes a context sequence and returns its result in distinct document order, whereas, for example, function `Descendants ()` in Linq consumes a single XML node and returns a list of element nodes in document order (without duplicate elimination)—we separate the XML path step traversal ($\texttt{/}\alpha\texttt{::}n$) from the call to the distinct document order function (`d-d-o`).

As mentioned in Section 2.2, we use algebraic placeholders to represent the XML functionality. This allows different back-ends to use different XML encodings. In Rule Doc column item is replaced by the result of the document lookup operator DOC.

$$
\frac{\Gamma; \mathit{loop} \vdash e \Mapsto q}{\Gamma; \mathit{loop} \vdash \texttt{doc}\ (e) \Mapsto \pi_{\mathsf{iter,pos,item:res}}\left(\mathrm{DOC}_{\mathsf{res:\langle item\rangle}}(q)\right)}\,(\textsc{Doc})
$$

$\texttt{/}\alpha\texttt{::}n$ consumes an arbitrary list of XML nodes and, for each context node, collects all accessible document nodes that match the structural predicate in $\alpha$ as well as the node test in $v$, and returns them in document order. Rule PathStep splits the path step semantics: operator $\lrcorner$ collects the result nodes and the ranking operator ⛉ adjusts the position information to reflect the surrogate order. As

$$
\begin{aligned}
\mathit{Expr} ::= {} & \ldots \\
& \mid \texttt{table}\ \mathit{ID}\ \texttt{(}\mathit{Integer}\,\texttt{,}\,\mathit{Integer}\texttt{)} \\
& \mid \texttt{tuple}\ \texttt{(}\mathit{Expr}\ (\texttt{,}\ \mathit{Expr})^{*}\texttt{)} \\
& \mid \mathit{Expr}\,\texttt{.}\,\mathit{Integer} \\
\ldots &
\end{aligned}
$$

Figure 2.8: Tuple support extending the grammar of basic LL (Figure 2.4).

discussed in Section 2.2, we expect the surrogate values to encode the document order.

$$\frac{\Gamma; \mathit{loop} \vdash e \Mapsto q}{\Gamma; \mathit{loop} \vdash e/\alpha\texttt{::}n \Mapsto \varrho_{\mathsf{pos:\langle item\rangle}}\left(\pi_{\mathsf{iter,item:res}}\left(\sqcup\!\!\lrcorner^{\alpha,n}_{\mathsf{res:\langle item\rangle}}(q)\right)\right)}\ (\textsc{PathStep})$$

For each iteration, Rule DistinctDocOrder eliminates all duplicate node surrogates ($\delta$) and adjusts the position information based on the document order ($\varrho$).

$$\frac{\Gamma; \mathit{loop} \vdash e \Mapsto q}{\Gamma; \mathit{loop} \vdash \texttt{d-d-o (}e\texttt{)} \Mapsto \varrho_{\mathsf{pos:\langle item\rangle}}\left(\delta\left(\pi_{\mathsf{iter,item}}(q)\right)\right)}\ (\textsc{DistinctDocOrder})$$

In Rule StringValue the node atomization operator ⚛ extracts the string values of the XML node list. While the underlying implementation of this operator, dependent on the context's node kind, may be quite complex—for example, the implementation of ⚛ for an element node needs to collect and merge the text nodes of the complete subtree—operator ⚛ always produces a single string for each context node.

$$\frac{\Gamma; \mathit{loop} \vdash e \Mapsto q}{\Gamma; \mathit{loop} \vdash \texttt{value (}e\texttt{)} \Mapsto \pi_{\mathsf{iter,pos,item:res}}\left(⚛_{\mathsf{res:\langle item\rangle}}(q)\right)}\ (\textsc{StringValue})$$

### 2.5.2 An Ordered View of Tables

So far, the compilation rules invariantly return operators with an iter|pos|item schema. The generated algebraic query plans store all their payload data in a single column item. If we want to query database-resident tables, construct tuples, and fetch the individual entries from a tuple and thus extend the basic LL grammar of Figure 2.4 with the grammar rules in Figure 2.8, we have to modify the invariant.

In a language where wider tables exist and hashes, named records, and tuples are built on top of it, we adjust our compilation scheme to replace the single $\mathsf{item}$ column by $n$ $\mathsf{item}$ columns ($\mathsf{item}_1, \ldots, \mathsf{item}_n$) where $n$ corresponds to the number of columns or entries a table or tuple provides. In consequence, our compilation rules then invariantly produce a $\mathsf{iter}|\mathsf{pos}|\mathsf{item}_1|\ldots|\mathsf{item}_n$ schema (with columns $\mathsf{iter}$ and $\mathsf{pos}$ still encoding the iteration and position information). Not all tuples and tables have the same width $n$—instead the width is prescribed by the number of $\mathsf{item}$ columns in the input plans.

To align all previous rules (Rules Const to StringValue) with the new invariant, (a) any projection argument on column $\mathsf{item}$ without renaming is replaced by all $n$ $\mathsf{item}$ columns visible in the input plans ($\pi_{\ldots,\mathsf{item}}$ is transformed into $\pi_{\ldots,\mathsf{item}_1,\ldots,\mathsf{item}_n}$) and (b) every remaining occurrence of column $\mathsf{item}$ is replaced by column $\mathsf{item}_1$.

Rule Table makes a database table $T$ with $n$ columns (and a key in the $i$-th column) accessible in LL. The compilation generates a table reference with $n$ $\mathsf{item}$ columns, whose position information is based on the key column (operator $\text{⛉}$). The cross product with *loop* establishes the correct loop lifted representation of the table.

$$\frac{q \equiv \mathit{loop} \times \left(\text{⛉}_{\mathsf{pos}:\langle \mathsf{item}_i \rangle} \left(\text{⛁}_{T\,(\mathsf{item}_1,\ldots,\mathsf{item}_n)}\right)\right)}{\Gamma; \mathit{loop} \vdash \texttt{table}\ T\texttt{(}n\texttt{,}i\texttt{)} \Rrightarrow q}\,(\textsc{Table})$$

In Rule Tuple all $n$ input expressions $e_i$ are compiled and merged into a single table. The query plans $q_1, \ldots, q_n$ are required to encode the same duplicate-free set of values in column $\mathsf{iter}$ to guarantee the correct alignment of the tuple values. The maintenance of $c_i$ and $c_{i\cdot sum}$ as well as the projection in $q_{i\cdot ext}$ ensures that no name conflicts between column names arise.

$$\frac{\begin{array}{l}
\Gamma; \mathit{loop} \vdash e_1 \Rrightarrow q_1 \\
c_1 \equiv |q_1.\mathit{cols} - \{\mathsf{iter}, \mathsf{pos}\}| \\
c_{1\cdot sum} \equiv c_1 \\
q_{1\cdot ext} \equiv q_1 \\
{\scriptstyle i=2,\ldots,n} \left|\begin{array}{l}
\Gamma; \mathit{loop} \vdash e_i \Rrightarrow q_i \\
c_i \equiv |q_i.\mathit{cols} - \{\mathsf{iter}, \mathsf{pos}\}| \\
c_{i\cdot sum} \equiv c_{(i-1)\cdot sum} + c_i \\
q_{i\cdot ext} \equiv q_{(i-1)\cdot ext} \overset{=}{\bowtie}_{\mathsf{iter}} \left( \pi_{\substack{\mathsf{iter},\mathsf{item}_{c_{(i-1)\cdot sum}+1}:\mathsf{item}_1, \\ \ldots,\mathsf{item}_{c_{(i-1)\cdot sum}+c_i}:\mathsf{item}_{c_i}}} (q_i) \right)
\end{array}\right.
\end{array}}{\Gamma; \mathit{loop} \vdash \texttt{tuple (}e_1, \ldots, e_n\texttt{)} \Rrightarrow q_{n\cdot ext}}\,(\textsc{Tuple})$$

Rule PosAccess accesses an individual table column or tuple component by projecting away all other item columns.

$$\frac{\Gamma; \mathit{loop} \vdash e \Rrightarrow q}{\Gamma; \mathit{loop} \vdash e\,.\,i \Rrightarrow \pi_{\mathsf{iter,pos,item_1:item}_i}(q)}\text{(PosAccess)}$$

### 2.5.3 A Uniform Variant of SQL/XML

The extended version of LL may be used to operate on the same data structures as SQL/XML [69], namely tables with XML columns. In contrast to SQL/XML, however, LL does not require two separate languages (SQL and XQuery) to query these data types.

Query $Q_3$ exemplifies the compilation, with extended order support, XML data, and tuple-aware rules with multiple item columns. The outcome is a table with an XML column.

```
for $op at $pos in d-d-o (doc ("algebra.xml")
                          /descendant::element(op))  (Q3)
return tuple ($pos, $op)
```

Figure 2.9 shows the corresponding query plan for Query $Q_3$ with annotations of intermediate results based on the XML document `algebra.xml` in Figure 2.10. We refer to the nodes of the XML document using the node surrogates $\gamma_0, \ldots, \gamma_7$. The evaluation of `doc`, for example, leads to $\gamma_0$—the root node surrogate.

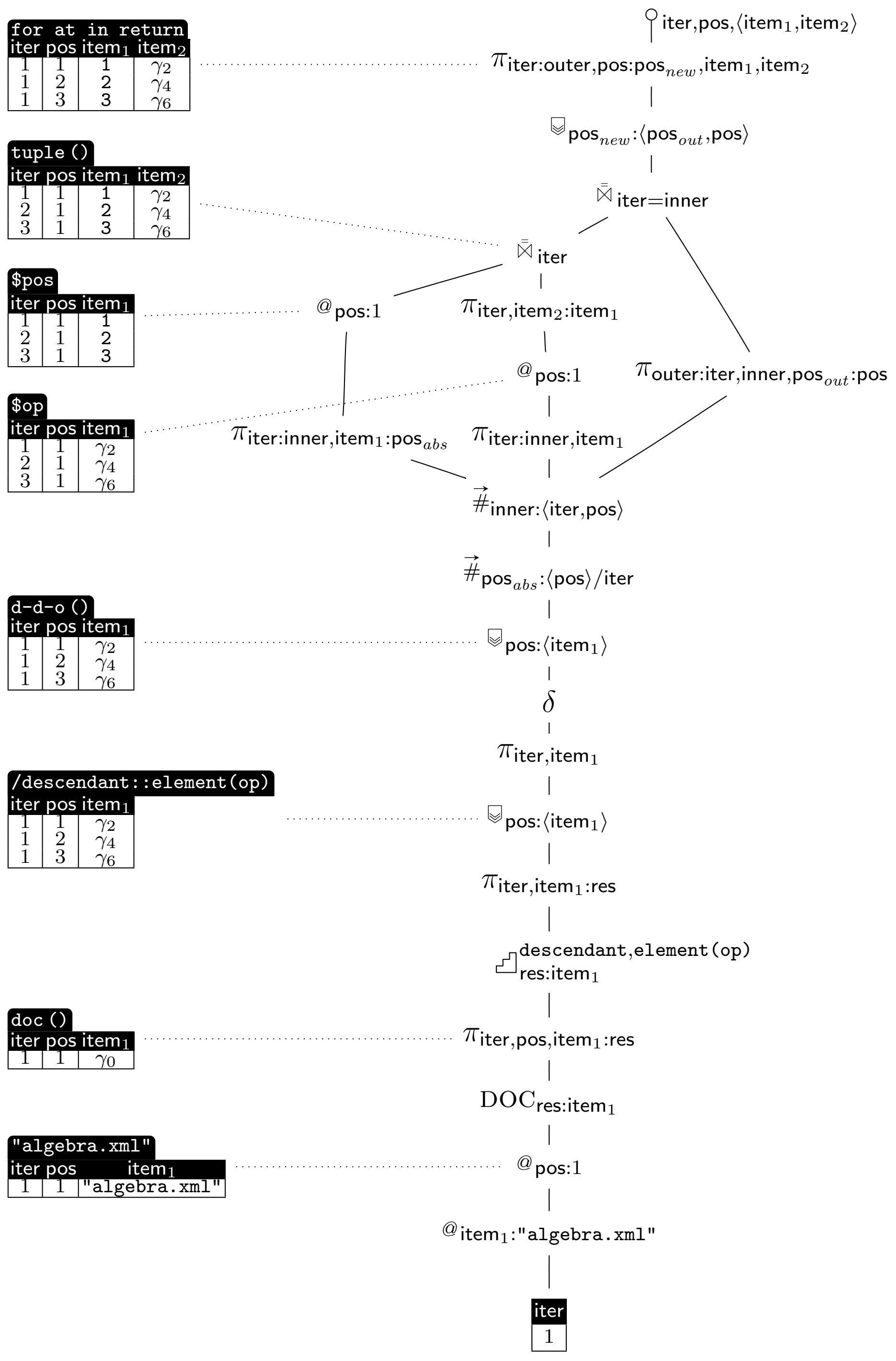
The `descendant` path step collects all reachable subtree element nodes with name `op` ($\gamma_2$, $\gamma_4$, $\gamma_6$). Based on these nodes, a new position column is calculated. As operator ⛉ is allowed to return values of an arbitrary domain, column pos in the intermediate result of the path step might simply feature the values $\gamma_2$, $\gamma_4$, and $\gamma_6$. The same is true for the upper rank operator describing the position information for the duplicate elimination (`d-d-o`). Operator $\vec{\#}_{\mathsf{pos}_{abs}:\langle\mathsf{pos}\rangle/\mathsf{iter}}$ enforces the generation of absolute position values for the position variable `$pos`.

Note how the resulting plan in Figure 2.9 features columns $\mathsf{item}_1$ and $\mathsf{item}_2$ to accommodate tuples. In the query plan the tuple constructor boils down to a single equi-join and a projection, which correctly renames the second item column.

## 2.6 A Flat Representation of Nesting

The data model of most companion languages allows arbitrary nesting of tuple and list constructors. With explicit list construction we extend LL to handle these nested data structures (Figure 2.11). Query $Q_4$ for example, generates a nested list that evaluates to `[[10,20],[30,40]]`.

```
list (list (10,20), list (30,40))  (Q4)
```



for at in return

| iter | pos | item$_1$ | item$_2$ |
|---|---|---|---|
| 1 | 1 | 1 | $\gamma_2$ |
| 1 | 2 | 2 | $\gamma_4$ |
| 1 | 3 | 3 | $\gamma_6$ |

tuple ()

| iter | pos | item$_1$ | item$_2$ |
|---|---|---|---|
| 1 | 1 | 1 | $\gamma_2$ |
| 2 | 1 | 2 | $\gamma_4$ |
| 3 | 1 | 3 | $\gamma_6$ |

$pos

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 3 |

$op

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | $\gamma_2$ |
| 2 | 1 | $\gamma_4$ |
| 3 | 1 | $\gamma_6$ |

d-d-o ()

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | $\gamma_2$ |
| 1 | 2 | $\gamma_4$ |
| 1 | 3 | $\gamma_6$ |

/descendant::element(op)

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | $\gamma_2$ |
| 1 | 2 | $\gamma_4$ |
| 1 | 3 | $\gamma_6$ |

doc ()

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | $\gamma_0$ |

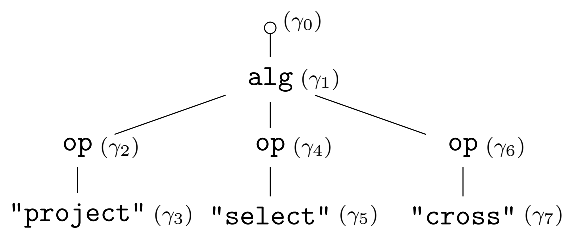
"algebra.xml"

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | "algebra.xml" |

Figure 2.9: Algebraic query plan encoding Query $Q_3$ with annotations. Tables on the left-hand side depict correspondence to LL query construct and intermediate result representation.

```
<alg>
  <op>project</op>
  <op>select</op>
  <op>cross</op>
</alg>
```

**(a)** `algebra.xml`.



**(b)** Tree representation of `algebra.xml`. ($\gamma_0, \ldots, \gamma_7$ depict node surrogate values.)

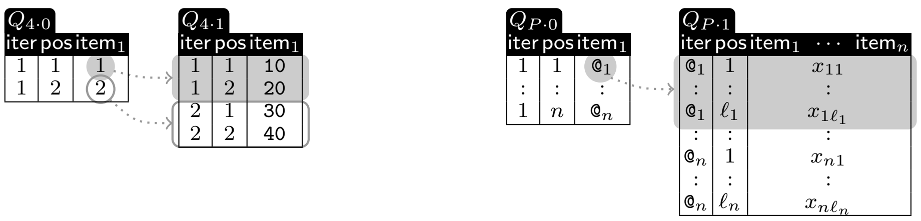
Figure 2.10: XML document `algebra.xml` in textual and tree representation.

$$
\begin{aligned}
\mathit{Expr} ::= {} & \ldots \\
& \mid \texttt{list}\,\texttt{(}\mathit{Expr}\ (\texttt{,}\ \mathit{Expr})^{*}\texttt{)} \\
& \mid \texttt{concat}\,\texttt{(}\mathit{Expr}\texttt{)} \\
& \mid \texttt{group}\,\texttt{(}\mathit{Expr}\,\texttt{,}\mathit{Expr}\texttt{)} \\
\ldots &
\end{aligned}
$$

Figure 2.11: Grammar extension of basic LL (Figure 2.4) to support nesting.

The LL extensions in Figure 2.11 furthermore provide means to flatten nested lists (via `concat`) and to group lists. The grouping primitive `group` positionally aligns its two arguments and subsequently groups the elements of the second argument by the corresponding grouping keys in the first argument.

We realize the nested data structures in our flat, first normal form algebra via surrogate (foreign key) values, variants of which have been used to realize non-first normal form database systems in the late 1980s [106]. For Query $Q_4$, for example, two query plans $Q_{4\cdot 0}$ and $Q_{4\cdot 1}$ together realize the nesting (Figure 2.12(a)): The outer query links to the inner query by means of the columns item$_1$ and iter, respectively.



$Q_{4\cdot 0}$

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 2 | 2 |

$Q_{4\cdot 1}$

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 10 |
| 1 | 2 | 20 |
| 2 | 1 | 30 |
| 2 | 2 | 40 |

$Q_{P\cdot 0}$

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | @$_1$ |
| : | : | : |
| 1 | $n$ | @$_n$ |

$Q_{P\cdot 1}$

| iter | pos | item$_1$ ··· item$_n$ |
|---|---|---|
| @$_1$ | 1 | $x_{11}$ |
| : | : | : |
| @$_1$ | $\ell_1$ | $x_{1\ell_1}$ |
| : | : | : |
| @$_n$ | 1 | $x_{n1}$ |
| : | : | : |
| @$_n$ | $\ell_n$ | $x_{n\ell_n}$ |

**(a)** Encoding the nested result of Query $Q_4$. **(b)** Encoding a generic nested list.

Figure 2.12: Relational runtime encoding of nesting on the database back-end. ($Q_{4\cdot 0}$ and $Q_{P\cdot 0}$ represent the outer lists; $Q_{4\cdot 1}$ and $Q_{P\cdot 1}$ all inner lists.)

In general, if a LL program produces a nested list value (of length $n$, $n \geqslant 1$) `[[`$x_{11}$`,`$x_{12}$`,...,`$x_{1\ell_1}$`],...,[`$x_{n1}$`,`$x_{n2}$`,...,`$x_{n\ell_n}$`]]`, the compiler forks the compilation process to translate the program into *two* separate relational queries:

1. one query that computes the relational encoding of the outer list `[`$@_1$`,...,` $@_n$`]` where all inner lists (including empty lists) are represented by surrogates $@_i$ ($Q_{P\cdot 0}$), and
2. one query that produces the encodings of *all* inner lists—assembled into a single table. If the $i$th inner list is empty, its surrogate $@_i$ will not appear in this table ($Q_{P\cdot 1}$).

Figure 2.12(b) depicts the resulting tabular encodings produced by the relational query pair. Note that the constituent queries are still flat queries to be evaluated over a first normal form database.

### 2.6.1 (Un)Boxing

The number of algebra plans for a given LL expression directly corresponds to its type: For every list type constructor $[\alpha]$, one query plan is generated. Query $Q_4$ for example, has a result type [[*integer*]] and thus is encoded by two query plans. Similarly Query $Q_5$—encapsulating Query $Q_4$—has result type [*integer*] and produces a single algebraic plan.

```
for $a in list(list(10,20), list(30,40))
return count($a)
```
($Q_5$)

But where does the information to split or merge the query plans come from? The compilation forks or fuses the query translation process through the insertion of calls to the helper functions `box` and `unbox`, respectively. The translation of function `box(`$e$`)` results in two query plans ($Q_{P\cdot 0}$ and $Q_{P\cdot 1}$) where the iteration context *loop* builds the basis for $Q_{P\cdot 0}$ and the relational representation of expression $e$ turns into $Q_{P\cdot 1}$. Similarly, function `unbox(`$e$`)` consumes two query plans and triggers the algebraic compiler to schedule a foreign key join between $Q_{P\cdot 0}$ and $Q_{P\cdot 1}$, which, effectively, dereferences the surrogates.

The introduction of `box` and `unbox` is performed by a simple, bottom-up static analysis phase that precedes algebraic compilation. This analysis—reminiscent of an inference of coarse types—is implemented by the set of inference rules shown in Figure 2.13. Judgment $e = e' : \tau$ inspects LL input subexpression $e$ to decide whether the relational encoding of the result of $e$ (a) consists of individual tuples of atomic values or surrogates ($\tau = \mathit{row}$), or (b) comes in tabular form ($\tau = \mathit{tbl}$). Given this, the analysis infers LL output expression $e'$, which features the required `box` and `unbox` invocations.

The following query depicts Query $Q_5$ after the static analysis phase:

```
for $a in list(box(list(10,20)), box(list(30,40)))
return count(unbox($a))
```

$$\frac{}{\mathit{val} = \mathit{val} : \mathit{row}}\,(1) \qquad \frac{}{\texttt{empty} = \texttt{empty} : \mathit{tbl}}\,(2) \qquad \frac{}{\texttt{\$v} = \texttt{\$v} : \mathit{row}}\,(3)$$

$$\frac{{}_{i=1,\ldots,n}\big|\; e_i = e_i' : \tau_i}{\texttt{append}\,(e_1,\ldots,e_n) = \texttt{append}\,(\boxed{\downarrow}^{\tau_1}_{\mathit{tbl}}(e_1'),\ldots,\boxed{\downarrow}^{\tau_n}_{\mathit{tbl}}(e_n')) : \mathit{tbl}}\,(4)$$

$$\frac{{}_{i=1,2}\big|\; e_i = e_i' : \tau_i}{(e_1 \circ e_2) = (\boxed{\downarrow}^{\tau_1}_{\mathit{row}}(e_1') \circ \boxed{\downarrow}^{\tau_2}_{\mathit{row}}(e_2')) : \mathit{row}}\,(5) \qquad \frac{e = e' : \tau}{\texttt{count}\,(e) = \texttt{count}\,(\boxed{\downarrow}^{\tau}_{\mathit{row}}(e')) : \mathit{row}}\,(6)$$

$$\frac{{}_{i=1,2,3}\big|\; e_i = e_i' : \tau_i}{\texttt{if}\,(e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3 = \texttt{if}\,(\boxed{\downarrow}^{\tau_1}_{\mathit{row}}(e_1))\ \texttt{then}\ \boxed{\downarrow}^{\tau_2}_{\mathit{row}}(e_2)\ \texttt{else}\ \boxed{\downarrow}^{\tau_3}_{\mathit{row}}(e_3) : \mathit{row}}\,(7)$$

$$\frac{e = e' : \tau}{\texttt{reverse}\,(e) = \texttt{reverse}\,(\boxed{\downarrow}^{\tau}_{\mathit{tbl}}(e')) : \mathit{tbl}}\,(8)$$

$$\frac{e = e' : \tau}{\texttt{unordered}\,(e) = \texttt{unordered}\,(\boxed{\downarrow}^{\tau}_{\mathit{tbl}}(e')) : \mathit{tbl}}\,(9)$$

$$\frac{{}_{i=\mathit{in},\mathit{return},1,\ldots,n}\big|\; e_i = e_i' : \tau_i}{\begin{array}{ll} \texttt{for \$v at \$p in}\ e_{\mathit{in}} & \texttt{for \$v at \$p in}\ \boxed{\downarrow}^{\tau_{\mathit{in}}}_{\mathit{tbl}}(e_{\mathit{in}}') \\ \texttt{order by}\ e_1,\ldots,e_n = & \texttt{order by}\ \boxed{\downarrow}^{\tau_1}_{\mathit{row}}(e_1'),\ldots,\boxed{\downarrow}^{\tau_n}_{\mathit{row}}(e_n') : \mathit{tbl} \\ \texttt{return}\ e_{\mathit{return}} & \texttt{return}\ \boxed{\downarrow}^{\tau_{\mathit{return}}}_{\mathit{row}}(e_{\mathit{return}}') \end{array}}\,(10)$$

$$\frac{e = e' : \tau}{\texttt{doc}\,(e) = \texttt{doc}\,(\boxed{\downarrow}^{\tau}_{\mathit{row}}(e')) : \mathit{row}}\,(11) \qquad \frac{e = e' : \tau}{e/\alpha\texttt{::}n = \boxed{\downarrow}^{\tau}_{\mathit{row}}(e')/\alpha\texttt{::}n : \mathit{tbl}}\,(12)$$

$$\frac{e = e' : \tau}{\texttt{d-d-o}\,(e) = \texttt{d-d-o}\,(\boxed{\downarrow}^{\tau}_{\mathit{tbl}}(e')) : \mathit{tbl}}\,(13) \qquad \frac{e = e' : \tau}{\texttt{value}\,(e) = \texttt{value}\,(\boxed{\downarrow}^{\tau}_{\mathit{row}}(e')) : \mathit{row}}\,(14)$$

$$\frac{}{\texttt{table}\ T(n,i) = \texttt{table}\ T(n,i) : \mathit{tbl}}\,(15) \qquad \frac{e = e' : \mathit{row}}{e.n = e'.n : \mathit{row}}\,(16)$$

$$\frac{{}_{i=1,\ldots,n}\big|\; e_i = e_i' : \tau_i}{\texttt{tuple}\,(e_1,\ldots,e_n) = \texttt{tuple}\,(\boxed{\downarrow}^{\tau_1}_{\mathit{row}}(e_1'),\ldots,\boxed{\downarrow}^{\tau_n}_{\mathit{row}}(e_n')) : \mathit{row}}\,(17)$$

$$\frac{{}_{i=1,\ldots,n}\big|\; e_i = e_i' : \tau_i}{\texttt{list}\,(e_1,\ldots,e_n) = \texttt{list}\,(\boxed{\downarrow}^{\tau_1}_{\mathit{row}}(e_1'),\ldots,\boxed{\downarrow}^{\tau_n}_{\mathit{row}}(e_n')) : \mathit{tbl}}\,(18)$$

$$\frac{e = e' : \tau}{\texttt{concat}\,(e) = \texttt{concat}\,(\boxed{\downarrow}^{\tau}_{\mathit{tbl}}(e')) : \mathit{tbl}}\,(19)$$

$$\frac{{}_{i=1,2}\big|\; e_i = e_i' : \tau_i}{\texttt{group}\,(e_1,e_2) = \texttt{group}\,(\boxed{\downarrow}^{\tau_1}_{\mathit{tbl}}(e_1'),\boxed{\downarrow}^{\tau_2}_{\mathit{tbl}}(e_2')) : \mathit{tbl}}\,(20)$$

Figure 2.13: Static analysis: introduction of `box()` and `unbox()` calls. The auxiliary function $\boxed{\downarrow}$ introduces `box` or `unbox` calls whenever the inference rule set implementing the judgment $e = e' : \tau$ encounters $\mathit{tbl}/\mathit{row}$ mismatches ($\boxed{\downarrow}^{\mathit{tbl}}_{\mathit{row}}(e) = \texttt{box}\,(e)$; $\boxed{\downarrow}^{\mathit{row}}_{\mathit{tbl}}(e) = \texttt{unbox}\,(e)$; $\boxed{\downarrow}^{\tau}_{\tau}(e) = e$).

As expression `list` (Boxing Rule 18) expects a tuple instead of a list, the static analysis resolves the $\llbracket\downarrow\rrbracket^{tbl}_{row}$ mismatch by introducing a call to `box`. Similarly, the aggregate `count` (Boxing Rule 6) requires a list, but initially consumes a tuple: `unbox` resolves this mismatch by dereferencing the nested lists.

Note that in a nested data model the semantics of the `for` loop has changed: Boxing Rule 10 expects tuples in $e_{return}$ and returns nested lists, if the return part of the `for` loop features lists.

## 2.6.2 Compilation of Nesting

To correctly reflect nesting in the inference rule set, we need to record the surrogate information that serves to relate the query plans. The adjusted compilation scheme defines the judgment

$$\Gamma; \mathit{loop} \vdash e \Mapsto (q, cs, ts) \ ,$$

which, unlike the original judgment defined in Section 2.3.1, returns a result *triple*. The enhanced compilation maintains the following central invariant: if $e$ is of type $[(t_1, \ldots, t_n)]$, plan $q$ will produce a table with schema $\mathsf{iter}|\mathsf{pos}|cs$ (with $cs = \mathsf{item}_1|\ldots|\mathsf{item}_n$), in which column $\mathsf{item}_i$ contains the values occurring at the $i$th tuple position. If $t_i$ is a list type $[\alpha_i]$, then $\mathsf{item}_i$ will be a column populated with surrogate values. In this case, the mapping $ts$ will contain an entry $\mathsf{item}_i \mapsto (q_i, cs_i, ts_i)$ that features an independent algebraic plan $q_i$, which computes the list contents.

The inference rules discussed so far can be categorized into two groups: rules that generate plans without nesting (Rules Const, Empty, Op, Count, and Doc–Table) and rules that ignore the plans in $ts$ and propagate them without modification (Rules Var, IfThenElse, For, OrderedFor, Reverse, Unordered, PosAccess, and Tuple). Rule Append forms an exception as merging nested lists requires special care to avoid conflicting surrogate values. Rule Append needs to introduce new unique surrogate values for all nested lists. We omit the details here and refer the interested reader to [109] for more details.

Rule List shares the compilation with Rule Append—its only distinction is the expected input type (compare Boxing Rules 4 and 18).

$$\frac{\Gamma; \mathit{loop} \vdash \texttt{append (}e_1\texttt{, ... ,}e_n\texttt{)} \Mapsto q}{\Gamma; \mathit{loop} \vdash \texttt{list (}e_1\texttt{, ... ,}e_n\texttt{)} \Mapsto q}\text{(List)}$$

Rule Concat consumes a list of lists and flattens it to produce a single list. The compilation of its input expression $e$ leads to a query plan with a single surrogate column $\mathsf{item}_1$. A foreign key join with query plan $q_{\mathsf{item}_1}$, describing the nested data, along the surrogate values leads to the expected flat representation.

$$\frac{\begin{array}{l}\Gamma; \mathit{loop} \vdash e \Mapsto (q, [\mathsf{item}_1], \{\mathsf{item}_1 \mapsto (q_{\mathsf{item}_1}, cs_{\mathsf{item}_1}, ts_{\mathsf{item}_1})\}) \\ q \equiv \varrho_{\mathsf{pos}_{new}:\langle \mathsf{pos}_{out},\mathsf{pos}\rangle}\left(\pi_{\mathsf{iter}_{out}:\mathsf{iter},\mathsf{iter}:\mathsf{item}_1,\mathsf{pos}_{out}:\mathsf{pos}}(q) \overset{=}{\bowtie}_{\mathsf{iter}} q_{\mathsf{item}_1}\right)\end{array}}{\Gamma; \mathit{loop} \vdash \texttt{concat (}e\texttt{)} \Mapsto \left(\pi_{\mathsf{iter}:\mathsf{iter}_{out},\mathsf{pos}:\mathsf{pos}_{new},cs_{\mathsf{item}_1}}(q), cs_{\mathsf{item}_1}, ts_{\mathsf{item}_1}\right)}\text{(Concat)}$$

The grouping operator `group` consumes two aligned lists where the first list describes the grouping keys and the second list describes the values to be grouped. Its result is a list of distinct group keys together with the corresponding lists of grouped values.

$$\frac{\begin{array}{l} {}_{i=1,2}\,|\,\Gamma; \mathit{loop} \vdash e_1 \Mapsto (q_i, cs_i, ts_i) \\ q_{1\cdot ext} \equiv \vec{\triangledown}_{\mathsf{grp}:\langle \mathsf{iter}, cs_1\rangle}(q_1) \\ \mathit{item} \equiv \mathsf{item}_{|cs_1|+1} \\ q_{grp} \equiv \delta\left(\pi_{\mathsf{iter},\mathsf{pos}:\mathsf{grp}, cs_1, \mathit{item}:\mathsf{grp}}(q_{1\cdot ext})\right) \\ q_{zip} \equiv \left(\vec{\#}_{\mathsf{pos}_{abs}:\langle \mathsf{pos}\rangle/\mathsf{iter}}(q_2)\right) \overset{=}{\bowtie}_{\mathsf{iter},\mathsf{pos}_{abs}} \left(\pi_{\mathsf{iter},\mathsf{pos}_{abs},\mathsf{grp}}\left(\vec{\#}_{\mathsf{pos}_{abs}:\langle \mathsf{pos}\rangle/\mathsf{iter}}(q_{1\cdot ext})\right)\right) \\ q \equiv \pi_{\mathsf{iter}:\mathsf{grp},\mathsf{pos},cs_2}(q_{zip}) \end{array}}{\Gamma; \mathit{loop} \vdash \texttt{group}\ \texttt{(}e_1\texttt{,}e_2\texttt{)} \Mapsto (q_{grp}, [cs_1, \mathit{item}], \{\mathit{item} \mapsto (q, cs_2, ts_2)\})} (\textsc{Group})$$

Rule GROUP creates the surrogate information for the nested lists by means of the row ranking operator $\vec{\triangledown}$. The created column (grp) becomes (a) the new surrogate column *item* for the grouped table ($q_{grp}$) and (b) the new iter column for the table storing the group values ($q$). A distinct operator $\delta$ performs the grouping for the outer table, whereas an equi-join on the iterations (column iter) and the absolute position values (column $\mathsf{pos}_{abs}$, calculated based on the relative positions in column pos) maps the surrogate values to the inner table ($q_{zip}$). Note how the result in Rule GROUP returns both query plans together with the correlation information (column *item*).

Rule BOX introduces a new plan based on the iteration context *loop* ($q_{P\cdot 0}$). As $q$ already encodes the matching iteration information in column iter, no additional code is required to create other surrogate values.

$$\frac{\begin{array}{c} \Gamma; \mathit{loop} \vdash e \Mapsto (q, cs, ts) \\ q_{P\cdot 0} \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter},\mathsf{item}_1:\mathsf{iter}}(\mathit{loop})\right) \end{array}}{\Gamma; \mathit{loop} \vdash \mathsf{box}\ (e) \Mapsto (q_{P\cdot 0}, [\mathsf{item}_1], \{\mathsf{item}_1 \mapsto (q, cs, ts)\})} (\textsc{Box})$$

To implement unboxing the inference rule has to apply the same foreign key join along the surrogate values as in Rule CONCAT. Rule UNBOX thus delegates the compilation to Rule CONCAT. The only minor difference is that `unbox` operates on a single tuple (instead of a list of tuples) and the first ordering criterion (column $\mathsf{pos}_{ord}$) of operator $\triangledown$ always provides the constant value `1`.

$$\frac{\Gamma; \mathit{loop} \vdash \texttt{concat (}e\texttt{)} \Mapsto q}{\Gamma; \mathit{loop} \vdash \texttt{unbox (}e\texttt{)} \Mapsto q} (\textsc{Unbox})$$

As calls to `box` and `unbox` cancel each other out (`unbox (box (`$e$`))` $\equiv e$), we expect the resulting query plans to be agnostic of the presence of superfluous `unbox`/`box` pairs. This is not true for the initial plans, as every call to Rule UNBOX leads to an additional join and a ranking operator. The optimizations in Chapter 4, however, ensure the above equivalence: The rewrites basically remove the joins and ranking operators, because the positions can be described by the old pos column and the join with *loop* will always deliver a single hit for each input row.

### 2.6.3 Nesting in Action

The following query presents Query $Q_4$ with boxing annotations after the static analysis phase:

```
list(box(list(10,20)), box(list(30,40)))
```

The query compiles into the algebra plans shown in Figure 2.14. The output features two query plans that describe the outer list as well as the nested lists. The evaluation of these plans leads to the result representation previously illustrated in Figure 2.12(a). Note that the outer lists can be compiled without any knowledge about the nested lists—the outer list depends on the initial *loop* relation $\begin{array}{|c|}\hline \text{iter} \\ \hline 1 \\ \hline\end{array}$ only.

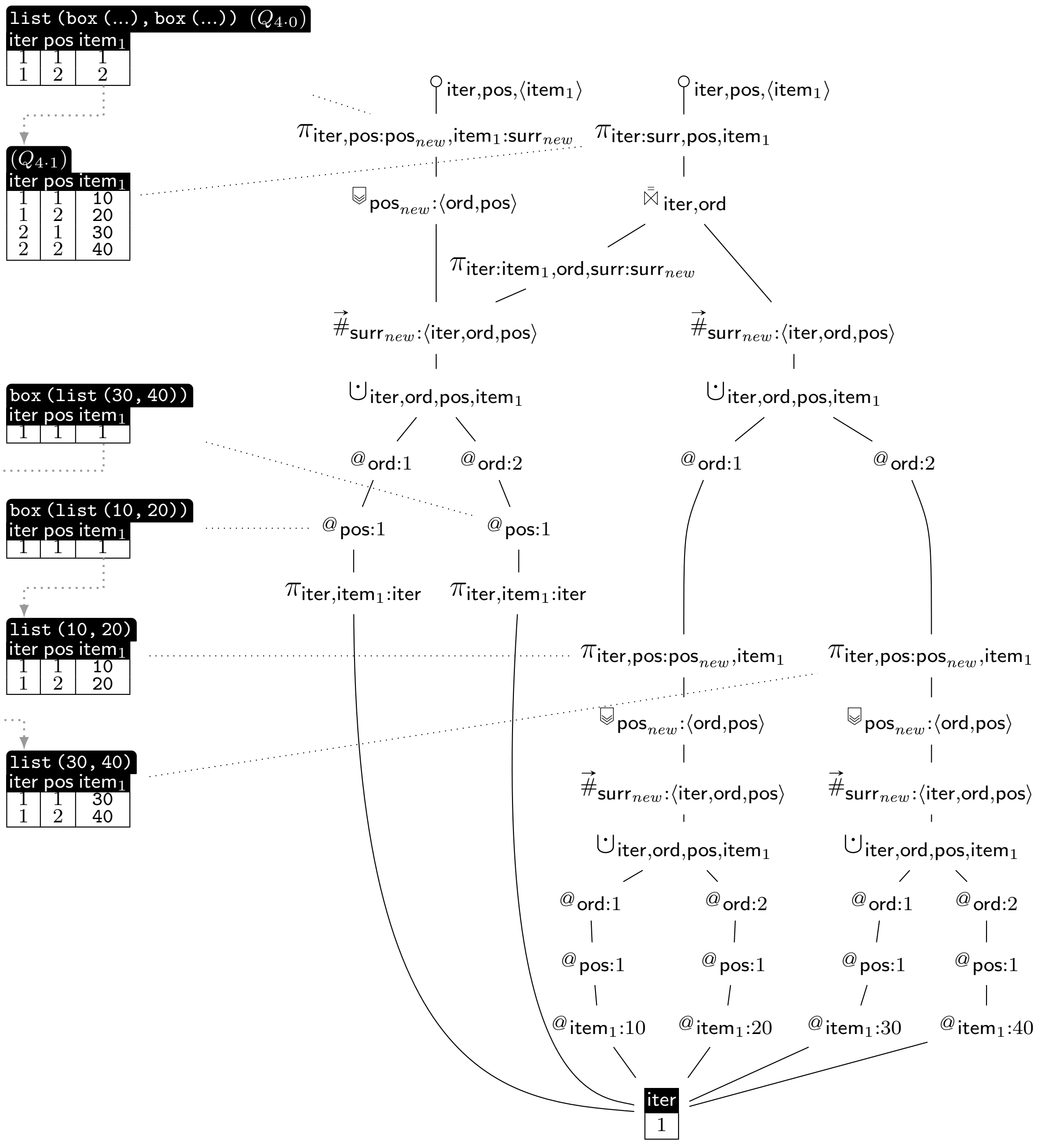
Previously, we mentioned the modification necessary to adjust Rule Append to the nested model. In Figure 2.14 we call the adjusted Rule Append via Rule List. The generated plan merges the input tables as well as the nested lists and replaces possibly conflicting surrogate values. The row numbering operators $\vec{\#}$ in Figure 2.14 introduce such new surrogate values, which are mapped to the nested lists if any exist, by means of an equi-join on the old surrogate value and column ord.

## 2.7 Summary and Related Work

Loop lifting was originally developed to directly compile XQuery expressions to SQL [63]. Follow-up work describes a loop lifted XQuery compilation strategy targeting a logical algebra [64]. Although this algebra is closely related to the algebra in Section 2.2, the operator semantics of some operators (e.g., ⊿) has slightly changed and new operators (e.g., the numbering operators $\vec{\Downarrow}$ and $\Downarrow$) were added. In this work XQuery's capabilities are expressed by LL and its order and XML extensions. Our compilation rules mainly differ from the original ones in [63, 64] in the relative encoding of sequence position information.

In the scope of the research prototype Ferry [42, 56]—a compiler for the purely functional language Ferry operating on database-resident tables—we extended the loop lifting compilation with tuples and nesting [109]. The tuple and nesting extensions of LL reflect these enhancements to the loop lifted compilation. LL and its order, tuple, and nesting extensions imitate Ferry's capabilities as well as the comprehensive core of the companion languages Links, Haskell, Ruby, and Python.

Linq operates on (possibly) nested records and allows to query both XML and table data. Taken together, the LL language constructs match the capabilities



list (box (...), box (...)) ($Q_{4\cdot 0}$)

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 2 | 2 |

($Q_{4\cdot 1}$)

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 10 |
| 1 | 2 | 20 |
| 2 | 1 | 30 |
| 2 | 2 | 40 |

box (list (30, 40))

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 1 |

box (list (10, 20))

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 1 |

list (10, 20)

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 10 |
| 1 | 2 | 20 |

list (30, 40)

| iter | pos | item$_1$ |
|---|---|---|
| 1 | 1 | 30 |
| 1 | 2 | 40 |

| iter |
|---|
| 1 |

Figure 2.14: Algebraic query plan encoding Query $Q_4$ with annotations. Tables on the left-hand side depict the corresponding LL query constructs as well as the intermediate results.

of LINQ's standard query operators. Microsoft's current LINQ implementation restricts the mixed evaluation of XML and table data in a single query: Database queries access table data, whereas XML data has to processed in the common language runtime, that is, on the programming language's in-memory heap. [115] tries to overcome this deficiency by storing XML data with an user-definable mapping in tables. An alternative approach transforms LINQ queries into a mix of XQUERY and SQL queries [116]. Our compilation approach offers a viable alternative to these approaches as we can target schema-less XML documents and do not have to bridge the gap between XQUERY and SQL [59]. The same observation might help to build a compiler for SQL/XML [69] without the "slash".

Database-supported program execution has come a long way from the first successful integrated database language Pascal/R [108] in the late 1970's to today's approaches like the ActiveRecord component of Ruby on Rails [101], Microsoft's LINQ to SQL provider [82], and the ADO.NET Entity Framework [39]. Loop lifting adds to this a viable alternative that is able to port more language concepts to a database back-end than any of these approaches [59, 110].

## 2.7.1 Logical Algebra and Order

The algebra described in Section 2.2 consists of RISC-like style operators. In most cases a large number of operators is necessary to, for example, represent the equivalent of a single SELECT box in IBM DB2's query graph model [99]. While the loop lifted compilation scheme introduces plan sharing in various places, which leads to DAG shaped query plans, the resulting plans tend to be more verbose than textbook style plans [38, 112]. A typical plan feature hundreds, not tens, of operators, but can be evaluated on a wide variety of back-end systems. Among others, this includes most SQL processors [55] and MonetDB/XQuery [21].[5]

The most important difference to other approaches, however, is our representation of order. Since the seminal work on interesting orders [111], order (and its propagation) is a concept located in the physical plan generation phase [38, 112]. In the presence of ordered domains—for example, item sequences in XQUERY—query compilers therefore directly generate physical plans [21] or introduce order-preserving logical algebras [24, 71, 114, 124]. The loop lifting approach, on the other hand, follows a different approach and represents order as data. Order is manipulated by means of the row numbering and row ranking operators $\vec{\#}$, $\vec{⊌}$, and $⊌$. This representation gives us the freedom to minimize order constraints independently of a particular back-end. Furthermore, these optimizations are orthogonal to *any* cost-based order optimization (such as [48, 91, 111]).

[5]More information on the code generators follows in Chapter 3.

### 2.7.2 Representing Nesting

The introduction of `box` and `unbox` by the compiler in Section 2.6.1 is inspired by techniques originally invented for the optimized translation of polymorphic primitives in programming languages [93]. Here, since we target relational database systems, we let `box` control the forking of the compilation process that emits bundles of SQL queries.

Note that there is a correspondence with the operators $\nu/\mu$ (or *nest*/*unnest*) introduced in the context of non-first normal form database systems [44, 106]. However, operators $\nu/\mu$ unfold their effect at *runtime* when a query is evaluated over a nested database, whereas `box`/`unbox` are *compile-time* concepts, which guide code generation for a flat first normal form relational database system.

Our use of the surrogates $\texttt{@}_i$ resembles an ordered variant of van den Bussche's approach to the simulation of the nested algebra via the flat relational algebra [122]. Whereas van den Bussche's work derives variable-width surrogate keys from the data itself, we opted to employ compact $\vec{\#}$- or $\vec{\text{⛊}}$-generated surrogates.

As of today, the elegance and conceptual beauty of the LINQ approach is not fully met by the LINQ provider implementations originally supplied by Microsoft. Whereas the current LINQ to SQL provider emits SQL statements and thus delegates query evaluation to a back-end database management system, all operations performed by the LINQ to XML provider are performed in-memory (on heap-resident representations of XML nodes). This mismatch of execution sites leads to asymmetries: in a nested LINQ query comprehension, the provider in charge for the *outermost* enclosing iteration "takes the lead".

If the LINQ to XML provider is in charge of the outer iteration, each iteration leads to the invocation of the LINQ to SQL provider. This mode of evaluation hits the relational database management system with an avalanche of (often similar) SQL queries, whose results have to be transferred into the heap for further evaluation by LINQ to XML (given that main-memory capacity suffices). In our setup it is exclusively a query's result *type*—not the database instance size—that determines the number of initiated database queries. The above mentioned query avalanche effect is avoided. This marks a significant deviation from Microsoft's original implementation and can have a profound performance impact [59].

### 2.7.3 More Loop Lifting

In Section 2.1 we started out with the basic idea of loop lifting and extended it with various language aspects. In this section we briefly sketch further additions. We skip the details of these extensions as they either do not affect the shape of the query plans or introduce new algebraic operators that are agnostic to most rewrites.

Although we added position information by means of column `pos` to represent ordered lists, we can also remove this column and thus understand unordered data. One use case for this scenario might be a loop lifted SQL compilation. Although nobody needs yet another SQL compiler, in the context of XQuery we have observed that loop lifting provides an excellent opportunity for declarative debugging [60]. A loop lifted SQL compilation thus provides especially good hooks for observations of intermediate expressions [58]. A similar extension enhances the loop lifted compiler with a new column type to support XQuery type matching [118]. Equivalently, XQuery fulltext scores [12] might be represented by a score column.

Additional relational placeholder operators allow us to faithfully integrate missing XQuery concepts such as node construction, runtime errors, and recursion. The compilation of XQuery's node constructors is described in [64] and [120]. Although side effects such as runtime errors contradict with the algebraic paradigm, we incorporated support for these errors by means of an additional query whose non-empty result indicates an error. Furthermore, instead of arbitrary recursion, we extended the loop lifting compiler with a controlled form or recursion, namely tail recursion [4, 5].

A large number of additional, yet straightforward inference rules is needed to extend LL with XQuery's function library [83], Linq's complete set of standard query operators [81], or XQuery 1.1's windowing functions [57]. Loop lifting allows us to express *any* positional constraint algebraically. Even the compilation of Linq's standard query operator `.Zip()`, performing positional alignment of (unrelated) lists, leads to a straightforward translation. Microsoft's Linq to SQL provider on the other hand fails to support this standard query operator as well as most other standard query operators involving sequence order [110].

Finally, loop lifting embraces higher-order functions (as proposed in the XQuery 1.1 Working Draft [104]). The loop lifted compilation of higher-order functions follows the same surrogate value strategy chosen for nested lists: A surrogate value representing a function item is generated when the function is defined and then dereferenced during function application (similar to calls of `box` and `unbox`). On application of a function, its function item—the surrogate value—is used to choose the correct expanded function body and to correctly loop lift a snapshot of all variables that were in scope at the function definition site, thus achieving the same effect as defunctionalization [102] in functional languages.

# Chapter 3

# Back-End Code Generation for Loop Lifted Query Plans

The loop lifted compilation described in the previous chapter leads to algebraic plans, whose characteristics differ significantly from the relational query plans found in database textbooks [38, 112]. The loop lifted algebra plans typically feature (a) a large overall number of operators with many numbering and mapping join operators, (b) plan sharing, and (c) XML placeholder operators.

To quantify these observations we compiled the 20 queries of the XMark benchmark [107]—a well-known benchmark in the XQuery domain—with Pathfinder. Table 3.1 lists the operator distribution averaged over all queries. An average XMark query contains around 35 numbering operators and 25 structural and mapping joins ($⏍$, $\overset{=}{\bowtie}$). Because of the loop lifted compilation, the query plans feature no value-based join—that is, joins comparing values of the XMark document. In total, the 20 query plans sum up to 5253 operators and 445 sharing points—operators with two or more parent operators.

In the following sections we discuss Pathfinder's two main back-end code generators as well as the evaluation of such algebra plans in the scope of the main-memory database system MonetDB/XQuery and in terms of SQL queries evaluated on top of IBM DB2.

## 3.1 MonetDB/XQuery

MonetDB/XQuery is the tight coupling of the relational main-memory database system MonetDB [18] and the XQuery compiler Pathfinder. MonetDB was designed to exploit modern CPU and memory hardware for query-intensive database management applications. We briefly sketch its main characteristics in the next section.

MonetDB/XQuery extends MonetDB in two places: It incorporates the compiler Pathfinder that emits its algebra plans in terms of MonetDB's query language and

| **Operator Category** | **Subsumed Operators** | **Average # of Operators** | **Fraction of Operators** |
|---|---|---:|---:|
| Projections | $\pi$ | 130.65 | 49.7% |
| Row Operators | $\sigma$, @, ⊚, CAST | 39.55 | 15.1% |
| Numbering Operators | $\vec{\#}$, $\vec{⛉}$, ⛉ | 35.25 | 13.4% |
| XML Operators | ⊿, DOC, ⚛, $\mathcal{N}$ | 25.70 | 9.8% |
| Mapping Joins | $\overset{=}{\bowtie}$ | 14.45 | 5.5% |
| Duplicate Elimination | $\delta$ | 6.75 | 2.6% |
| Set Operators | $\dot\cup$, $\setminus$ | 6.40 | 2.4% |
| Value-Based Joins | $\bowtie$, $\times$ | 0.00 | 0.0% |
| Others | $\boxed{a_1 \dots a_n}$, $\overset{name}{\boxed{a_1 \mid \dots \mid a_n}}$, ♀, GRP | 3.90 | 1.5% |
| Overall | | 262.65 | 100.0% |

Table 3.1: Categorized operator distribution for the average XMark query. ($\mathcal{N}$ represents node constructors.)

extends MonetDB's runtime with XML support. The runtime extensions feature, among others, XML shredding and serialization functionality, highly tuned (loop lifted) implementations of the Staircase Join path step algorithms [21, 66], and efficient implementations of XML node constructors, XML document lookup, and node atomization.

## 3.1.1 MonetDB

MonetDB is an open-source relational database system. Queries in MonetDB Version 4 are expressed in the *MonetDB Interpreter Language* (MIL) operating on an ordered binary table algebra [22]. This decomposed storage model [32, 78] is backed by *Binary Association Tables* (BATs) with exactly two columns: head and tail, whose storage structure are two aligned arrays of values.

MonetDB encodes a table with $n$ columns by $n$ BATs, in which all head columns are of of type *void*. A column of type *void* contains virtual object identifier values. Such a column stores only a single base object identifier *offset*, to represent a list of dense and ascending integer values ($\mathit{offset}, \mathit{offset}{+}1, \ldots, \mathit{offset}{+}n$). A foreign key join along *void* columns leads to positional array lookups (shifted by the value of the base object identifier *offset*).

The vertical fragmentation improves queries with a small number of columns as sequential scans of a few columns greatly reduce the main-memory access costs (by making optimal use of the CPU cache lines). This observation furthermore led to a fully materialized processing model where the processing of an operator only starts once all of its input rows are provided. Query processing algorithms, tuned to exploit the capabilities of modern CPUs, support this processing model.

Query optimization in MonetDB Version 4 consists of strategic and tactical query optimization. The strategic optimization is performed by the front-end that determines a good order of the operations in a query plan. The tactical optimization then chooses at runtime, based on runtime table characteristics, the best processing algorithm for a given operator.

## 3.1.2 MonetDB Interpreter Language

The interface language in MonetDB Version 4 is MIL—a procedural language extended with functions that represent the operators of the relational algebra such as `join()`, `select()`, and `sort()` as well as functions that manipulate the binary table model (e.g., `reverse()` and `mark()`). The algebra functions follow the characteristics of standard algebra operators—`join(`*a*`,`*b*`)` for example, performs an equi-join between the inner columns (tail of *a* and head of *b*) and returns a BAT where the join columns are projected out (resulting in the head of *a* and the tail of *b*). `reverse(`*a*`)` switches head and tail columns of a BAT *a* and `mark(`*a*`,`*offset*`)` replaces the tail column of *a* with a new dense key column of type *void* that starts from the object identifier *offset*.

The example MIL program in Figure 3.1(a) demonstrates the main characteristics of the resulting programs. The program creates a table with columns `a` and `b`, sorts this table by (`a`, `b`) and generates a new order column `c`. The program then filters the rows where `b` equals 1, projects onto column `a`, `c`, and `a`'s square (in column `f`), and finally serializes the table with columns `a` and `f`. The intermediate results for each of the variable assignment in Figure 3.1(a) are shown in Figures 3.1(b) to 3.1(o). Missing frames indicate columns of type *void*, which are not materialized.

The main observations with respect to Pathfinder's MIL code generation are: (a) Secondary orderings can be expressed by a concatenation of calls to `sort` and `CTrefine` (lines 5–6); (b) The result of `CTrefine`—an ordered grouping—can be also used to implement the operators ⩔ and $\delta$ (line 9), (c) Columns are aligned with a new order or a selection (lines 7, 11–12, and 15-19) by means of an order-preserving `leftjoin`; (d) The materialization of intermediate results is used to exploit plan sharing—column `a` for example, is scanned twice by the implicit join on the head columns, before the values within each matching row are multiplied (line 21).

## 3.1.3 Plan Generation for MonetDB

Pathfinder, as a front-end of MonetDB, performs a limited form of strategic optimizations. The logical query plans are transformed into plans expressed in an intermediate physical algebra over *n*-ary tables in which, for example, the numbering operators are replaced by explicit (*n*-ary) sorting and physical numbering operators such as `mark`. The physical algebra features a number of

```
 1  # fill literal table
 2  a := bat(void,int).seqbase(0@0).append(4).append(3).append(3);
 3  b := bat(void,int).seqbase(0@0).append(1).append(1).append(2);
 4  # sort by a then by b
 5  v1 := a.reverse().sort().reverse();
 6  v2 := v1.CTrefine(b)
 7  v3 := v2.mark(0@0).reverse();
 8  # assign the order extend to c
 9  c := v2.reverse().mark(0@0).reverse();
10  # adjust old columns to the new order with an order-preserving join
11  a := v3.leftjoin(a);
12  b := v3.leftjoin(b);
13  # select b=1
14  v4 := b.select(1);
15  v5 := v4.mark(0@0).reverse();
16  # apply filter to the columns with an order-preserving join
17  a := v5.leftjoin(a);
18  b := v5.leftjoin(b);
19  c := v5.leftjoin(c);
20  # project onto a, f:=a*a
21  f := [*](a, a);
22  # mark columns for garbage collection
23  b := nil;
24  # serialize a, f
25  print(a, f);
```

**(a)** Example MIL program.

a

| head | tail |
|---|---|
| 0@0 | 4 |
| 1@0 | 4 |
| 2@0 | 3 |

**(b)** Line 2.

b

| head | tail |
|---|---|
| 0@0 | 1 |
| 1@0 | 1 |
| 2@0 | 2 |

**(c)** Line 3.

v1

| head | tail |
|---|---|
| 2@0 | 3 |
| 1@0 | 3 |
| 0@0 | 4 |

**(d)** Line 5.

v2

| head | tail |
|---|---|
| 1@0 | 1@0 |
| 2@0 | 2@0 |
| 0@0 | 3@0 |

**(e)** Line 6.

v3

| head | tail |
|---|---|
| 0@0 | 1@0 |
| 1@0 | 2@0 |
| 2@0 | 0@0 |

**(f)** Line 7.

c

| head | tail |
|---|---|
| 0@0 | 1@0 |
| 1@0 | 2@0 |
| 2@0 | 3@0 |

**(g)** Line 9.

a

| head | tail |
|---|---|
| 0@0 | 3 |
| 1@0 | 3 |
| 2@0 | 4 |

**(h)** Line 11.

b

| head | tail |
|---|---|
| 0@0 | 1 |
| 1@0 | 2 |
| 2@0 | 1 |

**(i)** Line 12.

v4

| head | tail |
|---|---|
| 0@0 | 1 |
| 2@0 | 1 |

**(j)** Line 14.

v5

| head | tail |
|---|---|
| 0@0 | 0@0 |
| 1@0 | 2@0 |

**(k)** Line 15.

a

| head | tail |
|---|---|
| 0@0 | 3 |
| 1@0 | 4 |

**(l)** Line 17.

b

| head | tail |
|---|---|
| 0@0 | 1 |
| 1@0 | 1 |

**(m)** Line 18.

c

| head | tail |
|---|---|
| 0@0 | 1@0 |
| 1@0 | 3@0 |

**(n)** Line 19.

f

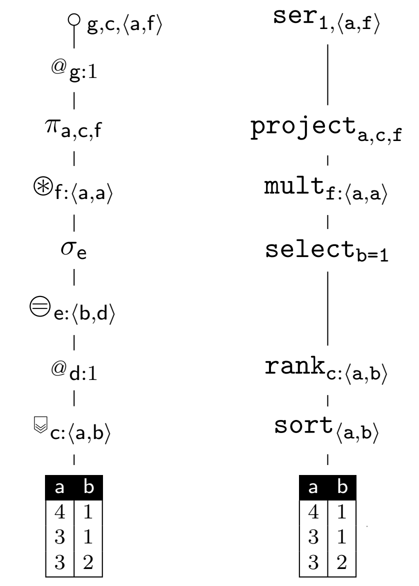
| head | tail |
|---|---|
| 0@0 | 9 |
| 1@0 | 16 |

**(o)** Line 21.

Figure 3.1: An example MIL program performing sorting, selection, and arithmetics (a) and its intermediate results (b)–(o). (The gray entries $x$@0 indicate columns of type *void*.)

important order-preserving operators, such as `leftjoin`, an $n$-ary `sort-refine` (similar to the MIL operator `CTrefine`), an order-aware `merge-union` operator, as well as the family of Staircase Join path step operators.

A cost model that severely punishes any sorting operator guides Pathfinder's plan enumeration. This leads to plans in which the row order is reused as often as possible (e.g., by means of `sort-refine`) and a minimum number of sorting operators remain to ensure the correctness of the query. An early version of MonetDB/XQuery, where the query plan was *always* ordered by columns (iter, pos), proved this fundamental decision a good choice [21].



**(a)** Logical query plan. **(b)** Physical query plan.

Figure 3.2: Logical query plan and its physical equivalent (resulting in the MIL program of Figure 3.1).

The second most important goal of the cost model is to choose plans in which duplicates are eliminated as early as possible. Duplicate elimination is especially cheap whenever a semi-join, a *general comparison join* [21, Section 4.2], or the physical path step operator Staircase Join [66] is chosen to prune duplicates on the fly. Furthermore, to avoid large intermediate results, additional duplicate elimination operators are introduced whenever appropriate.[1]

The rewrite rules of Pathfinder's plan generator are rather limited. Although more complex selection operators—read: standard algebraic selections—are formed, no join-reordering is applied. Therefore the shape of the resulting query plans always stays similar to their logical equivalents.

Figure 3.2 shows the logical query plan and its corresponding physical counterpart (leading to the MIL program in Figure 3.1). A rank operator ⩡ is transformed into a `sort` and a `rank` operator. The physical `select` operator replaces the three logical operators @, ⊜, and $\sigma$. Whereas the shape of the query plan stays the same, the physical plan now operates on an ordered data model. The plan enumeration enforces the correct output order and thus avoids the necessity to sort on the ordering criterion (column `c`) in the serialization point `ser`.

[1]Duplicate elimination is performed, whenever the input contains no key candidate and the removal of duplicates does not affect the overall result. Properties *key* and *set* discussed in Chapter 4 ease the detection of such situations.

### 3.1.4 MIL Code Generation

The MIL code generation can be described as a straightforward translation. A bottom-up, cost-based pattern matcher with uniform costs for any pattern chooses the rules that translate a group of one or more physical operators into a sequence of MIL assignments.[2] This translation transforms $n$-ary tables into $n$ BATs (as in Figure 3.1). Whenever a physical operator modifies the order or cardinality of its inputs, a batch of MIL `leftjoin`s align the affected BATs.

The translation of the physical query plan in Figure 3.2(b), for example, treats the combination of `rank` and `sort` as a single pattern and thus issues six translation rules in total. Figure 3.1 depicts the result of this translation.

In case a physical plan fragment is referenced more than once, because of the DAG structure of the query plans, its corresponding set of MIL variables, representing the $n$-ary output table, is reused.

#### Dead Code Elimination

Before the resulting MIL program is executed on MonetDB, an additional program analysis is performed. A dead code elimination analysis starts out from the side-effecting operations (e.g., `print` in Figure 3.1) and removes all assignments to unreferenced variables. The dead code elimination in the MIL program of Figure 3.1, for example, removes the generation and maintenance of column `c` (lines 9 and 19) and the last adaptation of column `b` (line 18). In a sense, the dead code elimination in MIL is a restricted form of projection pushdown known from logical query optimization [72].

## 3.2 SQL Code Generation

Pathfinder's SQL code generator does not target a specific back-end, but a wide range of possible relational back-ends. The code generator consumes a logical algebra plan and produces an equivalent SQL:1999 query. While we lose control over the execution of the generated SQL code, we gain extensive database support—for example, first-class cost-based query optimizers.

The SQL code generation relies on support for common table expressions (`WITH`-clauses) and SQL:1999's `ROW_NUMBER` and `DENSE_RANK` numbering functions to represent the DAG structure of the query plans as well as the numbering operators.

Regarding the XML-specific operators, we can make use of any relational XML encoding [45, 67, 80, 95], as long as the semantics of the XML placeholder operators are guaranteed. On IBM DB2, Microsoft SQL Server, and PostgreSQL we successfully applied various different encodings, for example, an edge-based

[2]The uniform costs ensure that the pattern matching favors larger patterns over single operators.

```
<open_auction id="1">
 <initial>
  15
 </initial>
 <bidder>
  <time>18:43</time>
  <increase>
   4.20
  </increase>
 </bidder>
</open_auction>
```

| pre | size | level | kind | name | value | data |
|---|---|---|---|---|---|---|
| 0 | 9 | 0 | DOC | au···xml | | |
| 1 | 8 | 1 | ELEM | open_··· | | |
| 2 | 0 | 2 | ATTR | id | 1 | 1.0 |
| 3 | 1 | 2 | ELEM | initial | 15 | 15.0 |
| 4 | 0 | 3 | TEXT | | 15 | 15.0 |
| 5 | 4 | 2 | ELEM | bidder | | |
| 6 | 1 | 3 | ELEM | time | 18:43 | |
| 7 | 0 | 4 | TEXT | | 18:43 | |
| 8 | 1 | 3 | ELEM | incre··· | 4.20 | 4.2 |
| 9 | 0 | 4 | TEXT | | 4.20 | 4.2 |

Figure 3.3: Encoding of the infoset of XML document `auction.xml`. Column data carries the nodes' typed decimal values.

pre/parent encoding [45] as well as region-based pre/post [67], pre/size/level [62], and ORDPATH encodings [95].

In the experiments performed in this work, DB2 operates on a table DOC, which stores an encoded version of persistent XML infosets [33]. We use a schema-oblivious node-based encoding of XML nodes in which, for each node $v$, key column pre holds $v$'s unique document order rank to form—together with columns size (number of nodes in subtree below $v$) and level (length of path from $v$ to its document root node)—an encoding of the XML tree structure (Figure 3.3 and [62]). XPath kind and name tests access columns kind and name—multiple occurrences of value `DOC` in column kind indicate that table DOC hosts several trees, distinguishable by their document URIs (in column name). For nodes with size $\leqslant 1$, table DOC supports value-based node access in terms of two columns that carry the node's untyped string value [37, § 3.5.2] and, if applicable, the result of a cast to type `xs:decimal`[3] (columns value and data, respectively).

## 3.2.1 Translating Algebra Plans to SQL

The SQL code generation consumes a logical query plan and partitions it into query blocks. The semantics of each such internal query block can be represented in terms of a single `SELECT·FROM·WHERE` expression. An internal query block consists of (a) a `FROM`-list collecting table references and literal table definitions, (b) a `SELECT`-list mapping logical column names to expressions, and (c) a `WHERE`-list storing conjunctive predicate expressions.

The partitioning algorithm that drives the translation starts at the plan leaves and greedily consumes operators until either an operator with multiple parent operators or one of the operators $\delta$, $\dot\cup$, $\setminus$, GRP, $\vec{\#}$, $\vec{\Cup}$, or $\Cup$ is reached.

Whenever the greedy operator consumption stops, the internal query block is

[3]In the interest of space, we omit a discussion of the numerous further XML Schema built-in data types.

```
WITH
-- binding because of rank operator
cte1 (a, b, c) AS
 (SELECT t1.a, t1.b,
         DENSE_RANK() OVER (ORDER BY t1.a ASC, t1.b ASC) as c
    FROM (VALUES (4, 1), (3, 1), (3, 2)) AS t1 (a, b))

SELECT 1 AS g, t2.c AS c, t2.a, t2.a * t2.a as f
  FROM cte1 as t2
 WHERE t2.b = 1
 ORDER BY g ASC, c ASC;
```

Figure 3.4: Generated SQL code for the logical query plan in Figure 3.2(a).

transformed into a `SELECT·FROM·WHERE` expression. This expression is extended (e.g., with an additional `GROUP BY` clause) to correctly reflect the semantics of the operator that stopped the consumption and forms a new common table expression. The partitioning continues with a name reference to the new common table expression and completes at the serialization point ♀ where the last internal query block is turned into the top-level `SELECT·FROM·WHERE` expression. For more information on Pathfinder's SQL code generation we refer the interested reader to [85].

Figure 3.4 shows the generated SQL code for the logical query plan in Figure 3.2(a). The blocking rank operator ⛉ stops the partitioning and leads to the common table expression bound to the name `cte1`. The internal query block of operator ⛉ is filled with references to the common table expression (`FROM`-list: `cte1`) and its corresponding column expressions for all three visible columns (`SELECT`-list: `a` → `a`, `b` → `b`, `c` → `c`).

Operators @ and ⊜ add two more entries to the `SELECT`-list (`d` → `1`, `e` → `b = 1`) and the translation of operator $\sigma$ copies the expression for column `e` from the `SELECT`-list to the `WHERE`-list (`b = 1`). At the end the serialization point prunes the `SELECT`-list, builds a `SELECT·FROM·WHERE` expression, and orders the result by the iteration and position values in columns `g` and `c`.

## 3.3 A First Quantitative Assessment

Both back-end code generators sketched in the previous sections, are built into Pathfinder. To get a first impression of the runtime characteristics, we compiled both MIL and SQL code for XMark Query 8 with Pathfinder and executed the resulting queries on MonetDB/XQuery v0.36 and DB2 V9.7 against XMark

| | **sf 0.01** | **sf 0.1** | **sf 1** |
|---|---|---|---|
| MonetDB/XQuery | 182 | 11,870 | 2,278,786 |
| Pathfinder + DB2 | 477 | 2,346,302 | – |

Table 3.2: Evaluation times of XMark Query 8 (in msec, averaged over 10 runs) with varying scale factors (sf) where sf 1 corresponds to an XML document of 110 MB size. (No optimizations have been performed yet.)

instances of scale factor 0.01, 0.1, and 1.[4]

XMark Query 8 features 11 XPath steps, two nested `for` loops, a predicate formulating a value-based join, three node constructors, and an aggregate. The query returns its result in the document order of the `person` nodes:

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

The loop lifted compilation of XMark Query 8 leads to a tall stacked algebra plan with 286 algebra operators that features a typical operator distribution matching the distribution of the average XMark query in Table 3.1 and 25 sharing points. The resulting MIL code amounts to 3282 lines of MIL text (107 KB) and the generated SQL query consists of 466 lines with 56 common table expressions (22 KB). DB2's execution plan for XMark Query 8 features 171 operators with 15 sort and 42 join operators.

Table 3.2 shows the evaluation times for the varying XML document sizes (ranging from 1.1 MB to 110 MB) both on MonetDB/XQuery and DB2. Although the query runs in interactive time on the smallest document, the query scales quadratically with the document size in MonetDB/XQuery and even worse on DB2. These slow evaluation times are not surprising, as the loop lifted query plans prescribe an evaluation strategy: For every `person` element, the execution plan of XMark Query 8 retrieves all `closed_auction` elements before the filter `$t/buyer/@person = $p/@id` is applied.

For other XMark queries MonetDB/XQuery, however, demonstrates that the prescribed evaluation order of the loop lifted algebra plans does not necessarily lead to bad execution plans. 13 out of the 20 XMark queries are executed within 100 milliseconds on scale factor 0.1 and in less than 1 second for scale factor 1.[5] These queries have in common that they do not generate large intermediate results.

[4]Peek forward to Section 5.2 for more details on the setup of the experiment.

[5]For more details peek forward to Table 5.2 in Chapter 5.

```
var b1 := bat(find(monet_guide,"site.people.person.id@"));
var b2 := bat(find(monet_guide,"site.people.person.name"));
var b3 := bat(find(monet_guide,"site.people.person.name.cdata"));
var b4 := bat(find(monet_guide,"site.people.person.name.cdata.string@"));
var b5 := bat(find(monet_guide,
                   "site.closed_auctions.closed_auction.buyer.person@"));
var b6 := outerjoin(b1,reverse(b5));
var b7 := join(join(b2,b3),b4);
var b8 := histogram(reverse(b6));
b8@batloop()
  { printf("<item person=\"%s\">%i</item>\n", find(b7,$h), $t); }
```

Figure 3.5: MIL formulation of XMark Query 8 in [107, System D].

Compared to the "baseline" measurements in the original XMark article [107], the results in Table 3.2 differ significantly: For XMark Query 8 on scale factor 1 the authors report less than 10 seconds evaluation time for the slowest system (on a slower machine). Interestingly, the fastest time reported in [107, Table 3]—470 msec—was measured on MonetDB. Figure 3.5 shows the MIL code used to observe the reported time. The MIL code uses an XML encoding based on *DataGuides* [47] and is the result of an hand-optimized translation of the XMark query semantics. The MIL query features calls to `outerjoin` and `histogram` and differs strikingly from the automatically generated query plans produced by the loop lifting compiler. Although this "baseline" comparison might be unfair, we accept the fact that more work is necessary to turn our compositional compilation of *arbitrary* non-relational queries into a competitive approach.

## 3.4 Summary

With the two back-end code generators in place, we can now turn a large number of database systems into query processors for non-relational query languages. The SQL code generator allows us to target *any* database system supporting the SQL:1999 dialect, while relying on their built-in optimization and index infrastructure. With MonetDB/XQuery, on the other side, we are able to drive a fast main-memory database system with extensive XML support.

Furthermore there are currently efforts underway to extend Pathfinder with back-end code generators for the next generation database systems MonetDB Version 5 [77] and X100 [17]. In general, any database system that supports the algebra dialect described in Section 2.2 might be turned into an evaluation back-end for non-relational queries. Because of the abstraction of the XML operators, this statement is also true for any XML-processing database system (e.g., native XML database systems such as Natix [43]).

The analysis in Section 3.3 revealed that both MonetDB/XQuery and DB2 were not able to change the implicit evaluation order prescribed by the loop lifted compilation scheme described in Chapter 2. For MonetDB/XQuery a more advanced strategic query optimizer and a better cost model would certainly improve the quality of the execution plans. In this setting, a good query optimizer would need to take into account (a) the DAG shape of the query plans, (b) the large number of non-traditional numbering operators, (c) a large number of mapping joins, and (d) MonetDB's intermediate result materialization strategy. When we analyzed the execution plans of DB2 and other SQL database systems it turned out that their optimizers, too, were overwhelmed by just these plan characteristics, namely (a) the common table expressions, (b) the SQL incarnations of the numbering operators—the window functions `ROW_NUMBER` and `DENSE_RANK`—and (c) the large number of mapping joins.

In what follows, we aim to improve the execution plans of *all* possible back-ends, instead of bringing one system to perfection. The optimizations discussed in Chapter 4 target the logical algebra plans and thus affect all back-ends.

# Chapter 4

# Logical Query Optimization for Loop Lifted Algebra Plans

For over 30 years database systems have now excelled in providing fast access to large quantities of data. Whereas logical query optimization in database textbooks is restricted to select-project-join queries [38, 112], most database systems take a much larger set of rewrites into account and apply more advanced rewrites such as decorrelation of nested subqueries. The query shape of the loop lifted algebra plans, however, overwhelmed any query optimizer we had on our workbench. For all back-ends the combination of plan size, plan sharing, numbering operators, and mapping joins resulted in execution plans whose evaluation strategy largely reflects the structure of the surface queries—for example, an execution plan calculates a Cartesian product and only afterwards applies a predicate to implement a surface query with two nested `for` loops and a filter expression.

Here, we take the characteristics of the loop lifted algebra plans into account and provide a set of *rewrite rules* that considerably simplify the logical query plans. Our goal is to provide plans with *join graphs*—algebra plans consisting of bundles of base table references and joins—and subsequent *plan tails* performing grouping, duplicate elimination, and ordering, whenever possible. Relational database systems are specifically tuned to cope with this class of queries. To generate such queries we optimize the query plans based on the following three *heuristics*:

- **House Cleaning** (Section 4.2). The rewrites that implement this heuristic prune unnecessary operators and reduce the operators' arguments: They remove plan fragments that are guaranteed to yield an empty table, simplify operators based on key and constant information, perform variants of selection and projection pushdown [72], and eliminate common subplans.

- **Order Minimization** (Section 4.3). The group of order rewrites are geared to exploit as much information as possible to simplify and remove order constraints—the row numbering and ranking operators—from the query plans.

- **Query Unnesting** (Section 4.4). Two distinct sets of rewrite rules relate to query unnesting. The first set removes the explicit column alignment—the mapping joins—and abridges the duplicate elimination. Based on the resulting straight-line chain of operators the second set of rules disentangles independent operator groups.

Before we delve into the heuristics' details, the following section gives an overview of the rewrite framework used to optimize the large DAG shaped algebra plans.

## 4.1 Peephole Optimization

A holistic query optimization approach does not fit well with the large number of operators and the plan sharing we are faced with. From the world of compiler construction we therefore adopt a local rewrite technique: peephole optimization [87]. Similar to local inspections of object code, the optimizer takes patterns consisting of only a few algebraic operators into account.

The peephole optimization is supported by a simple form of plan analysis that detects a variety of column and operator properties. Based on these properties interesting operators—matching the conditions of a rewrite rule—are spotted and become subject to simplification. A property is synthesized in either a single *top-down* or a single *bottom-up* walk of the DAG. As a side effect of the property inference each plan operator is annotated with with the property information.

The rewrite framework that performs the optimizations is not limited to any number of rewrite rules or properties. The set of rewrite rules described in this work is grouped into sensible units—heuristics—and on a more fine-grained level into *rule classes*, similar to the rule engine in [99]. Each rule class relies on a number of plan properties. Here, we make use of 12 different properties, but additional properties may improve the property inference of existing properties and provide further information for rewrites.

### Property Inference and Rewrite Notation

The rewrites take various properties into account. These properties collect information on keys (*key*), constant columns (*const*) that store the same value in all rows, abstract domain relationships (*dom*), functional dependencies (*fd*), available as well as required columns (*cols*, *icols*), the use of columns (*use*), expected selection values (*req*), duplicates (*set*), operator cardinality (*empty* and *card1*), and the number of parent operators (*#ref*). We present each property in more detail at its first point of reference (either by a rewrite or another property).

| **Column Property** *cols* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}\,(q)$ | $\mathit{cols} \leftarrow \{a_1, \dots, a_n\}$ |
| $\sigma_b\,(q)$ | $\mathit{cols} \leftarrow q.\mathit{cols}$ |
| $\times\,(q_1, q_2)$, $\overset{\equiv}{\bowtie}_{b_1=b_2}\,(q_1, q_2)$, $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1, q_2)$ | $\mathit{cols} \leftarrow q_1.\mathit{cols} \,\cup\, q_2.\mathit{cols}$ |
| $\delta\,(q)$ | $\mathit{cols} \leftarrow q.\mathit{cols}$ |
| $\dot\cup_{b_1,\dots,b_n}\,(q_1, q_2)$, $\setminus_{b_1,\dots,b_n}\,(q_1, q_2)$ | $\mathit{cols} \leftarrow \{b_1, \dots, b_n\}$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\dots,a_n:\circ_n(b_n)/b_{grp}}\,(q)$ | $\mathit{cols} \leftarrow \{b_{grp}, a_1, \dots, a_n\}$ |
| $\boxed{\begin{smallmatrix} a_1 \dots a_n \\ \varnothing \end{smallmatrix}}$, $\boxed{\begin{smallmatrix} a_1 \dots a_n \\ \square\square\square \end{smallmatrix}}$, $\boxed{\begin{smallmatrix} \mathit{name} \\ a_1 \,\dots\, a_n \end{smallmatrix}}$ | $\mathit{cols} \leftarrow \{a_1, \dots, a_n\}$ |
| $\textcircled{r}_{a:\langle b_1,\dots,b_n\rangle}\,(q)$, $\vec{\#}_{a:\langle b_1,\dots,b_n\rangle/b_{grp}}\,(q)$, $\vec{\triangledown}_{a:\langle b_1,\dots,b_n\rangle}\,(q)$, $\triangledown_{a:\langle b_1,\dots,b_n\rangle}\,(q)$, $\llcorner^{\alpha,v}_{a:\langle b\rangle}\,(q)$ | $\mathit{cols} \leftarrow q.\mathit{cols} \,\cup\, \{a\}$ |
| $\text{⚲}_{b_1,b_2,\langle b_3,\dots,b_n\rangle}\,(q)$ | $\mathit{cols} \leftarrow \{b_1, \dots, b_n\}$ |

Table 4.1: Bottom-up inference of the column property *cols* that collects the set of visible columns.

To keep the rewrite rules as well as the property inference notation compact, we replace the algebra operators $@_{a:val}$, $\textcircled{r}_{a:\langle b_1,\dots,b_n\rangle}$, $\mathrm{CAST}^{type}_{a:\langle b\rangle}$, $\mathrm{DOC}_{a:\langle b\rangle}$, and $\text{⚛}_{a:\langle b\rangle}$ in the following by a generic row-based operator $\textcircled{r}_{a:\langle b_1,\dots,b_n\rangle}$. These five operators share their operator characteristics for most properties and rewrites.

Table 4.1 denotes the inference of the column property *cols* that collects the emitted columns for each operator. The property is inferred bottom-up. The left-hand side of Table 4.1 indicates the affected operators and their arguments. The right-hand side shows the corresponding property inference rule. For example, for the projection operator $\pi$ the right-hand side reads: "Assign to *cols*—the column property *cols* of the current operator—the set of columns $a_1, \dots, a_n$", thus matching the columns of the projection argument. $q.\mathit{cols}$ similarly refers to the column property of the child operator $q$ and $\cup$ indicates the set union of two sets of properties.

All other properties follow the same notation as the column property in Table 4.1. The following property inference rule, for example, stems from the derivation of the constant property (Table 4.3):

$$\pi_{a_1:b_1,..,a_n:b_n}\,(q) \qquad\qquad \mathit{const} \leftarrow \{(a_i, v) \,|\, (b_i, v) \in q.\mathit{const}\}$$

Based on the constant information of operator $q$, the rule performs a lookup for all $(\mathit{name}, \mathit{value})$ pairs that match an input column $b_1, \dots, b_n$ and binds the corresponding constant value to $v$. For each match a new entry $(a_i, v)$, where $i$ corresponds to the respective index of $b_i$, is generated. As this form of notation heavily relies on pattern matching, it helps us to keep the notation of the inference rules compact. Additionally, we share the property inference of multiple operators, whenever possible (such as operators $\times$, $\bar{\bar{\bowtie}}$, and $\bowtie$ in Table 4.1).

Some property inference phases make use of additional information on constants (CONST, CONST-VAL), keys (KEY), and cardinality (CARD), whenever the information is available for literal and database tables.[1]

Rewrites are described using an inference rule notation where an operator (or a pair of operators) on the left-hand side is rewritten into a different plan if all conditions in the premise are fulfilled. The following example shows a typical rewrite rule where a projection $\pi$ is removed—resulting in the input plan $q$—whenever the set of available columns in $q$ is identical to the set of projection columns $\{a_1, \ldots, a_n\}$:

$$\frac{\{a_1, \ldots, a_n\} = q.cols}{\pi_{a_1,\ldots,a_n}(q) \rightarrow q} (4)$$

As for the properties, we make use of a form of pattern matching to keep the rewrite rules compact. In addition to the set comparison $(\{\ldots\} = \ldots)$, we use $\in$ and $\subseteq$ to check for the existence of one and multiple items in a set, respectively.

## 4.2 House Cleaning

The first heuristic aims to perform house cleaning. To achieve this effect, the heuristic prunes plan fragment that yield an empty table (Section 4.2.1), applies common simplifications (Section 4.2.2), incorporates variants of the well-known selection and projection pushdown heuristics (Section 4.2.3 and Section 4.2.4), and detects and eliminates common subplans (Section 4.2.5).

### 4.2.1 Empty Table Propagation

The first rule class removes plan fragments that evaluate to an empty table. The roots of these subplans are statically detected by the empty-input property *empty* (Table 4.2), which propagates information on empty literal tables up in the plan. Possible values of *empty* are the Boolean values '*true* and '*false*.

The following three rewrite rules are capable to remove *all* possible subplans that yield an empty table detected by the empty-input property inference. Instead of replacing all combinations of operators that reference an empty literal table by an empty literal table, the rewrites analyze only the places where the value of *empty* might change and remove the subplans that are guaranteed to not emit any result row.

$$\frac{q_1.empty}{q_1 \dot{\cup}_{b_1,\ldots,b_n} q_2 \rightarrow \pi_{b_1,\ldots,b_n} q_2} (1) \qquad \frac{q_2.empty \qquad \circledast \in \{\dot{\cup}, \setminus\}}{q_1 \circledast_{b_1,\ldots,b_n} q_2 \rightarrow \pi_{b_1,\ldots,b_n} q_1} (2)$$

[1] The unavailability of this information does not break the property inference phases, but might lead to less applications of rewrites.

| **Empty-Input Property** *empty* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}(q)$, $\sigma_b(q)$ | $empty \leftarrow q.empty$ |
| $\times(q_1,q_2)$, $\overset{=}{\bowtie}_{b_1=b_2}(q_1,q_2)$, $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}(q_1,q_2)$ | $empty \leftarrow q_1.empty \vee q_2.empty$ |
| $\delta(q)$ | $empty \leftarrow q.empty$ |
| $\dot\cup_{b_1,\ldots,b_n}(q_1,q_2)$ | $empty \leftarrow q_1.empty \wedge q_2.empty$ |
| $\setminus_{b_1,\ldots,b_n}(q_1,q_2)$ | $empty \leftarrow q_1.empty$ |
| $\text{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}(q)$ | $empty \leftarrow q.empty$ |
| $\begin{array}{\|c\|}\hline a_1 \ldots a_n \\ \hline \varnothing \\ \hline\end{array}$ | $empty \leftarrow$ ‘*true* |
| $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|c\|}{a_1 \ldots a_n} \\ \hline & & \\ \hline\end{array}$, $\begin{array}{\|c\|c\|c\|}\multicolumn{3}{c}{name} \\ \hline a_1 & \ldots & a_n \\ \hline\end{array}$ | $empty \leftarrow$ ‘*false* |
| $\textcircled{r}_{a:\langle b_1,\ldots,b_n\rangle}(q)$, $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(q)$, $\vec{⩔}_{a:\langle b_1,\ldots,b_n\rangle}(q)$, $⩔_{a:\langle b_1,\ldots,b_n\rangle}(q)$, $⌟^{\alpha,v}_{a:\langle b\rangle}(q)$, $♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}(q)$ | $empty \leftarrow q.empty$ |

Table 4.2: Bottom-up inference of the empty-input property *empty* that derives whether the evaluation of an operator produces no rows at runtime.

$$\frac{q.empty \qquad q \notin \{\begin{array}{|c|}\hline a_1 \ldots a_n \\ \hline \varnothing \\ \hline\end{array}\}}{♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}(q) \rightarrow ♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}(\begin{array}{|c|}\hline b_1 \ldots b_n \\ \hline \varnothing \\ \hline\end{array})}\,(3)$$

For the query plan of Query $Q_2$ (Figure 2.5; page 16), for example, Rewrite 2 removes the top-most union operator together with its right-hand side subplan—the rewrite effectively prunes the `else` branch of Query $Q_2$.

## 4.2.2 Standard Operator Simplification

All simplifications proposed in this section operate on standard operators available in every relational database system. Because most rewrites require more context information than currently available, we will interleave the discussion of the rewrites with the introduction of new properties.

Rewrites 4 and 5 both remove superfluous projection operators that either do not affect the input $q$ or are superseded by a parent projection.

$$\frac{\{a_1,\ldots,a_n\} = q.cols}{\pi_{a_1,\ldots,a_n}(q) \rightarrow q}\,(4) \qquad \frac{}{\pi_{a_1,\ldots,a_n}(\pi_{b_1,\ldots,b_m}(q)) \rightarrow \pi_{a_1,\ldots,a_n}(q)}\,(5)$$

The conditions of Rewrites 6 to 10 depend on the constant property *const* defined in Table 4.3. The constant property inference collects pairs of columns

| **Constant Property** *const* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}\,(q)$ | $const \leftarrow \{(a_i, v) \mid (b_i, v) \in q.const\}$ |
| $\sigma_b\,(q)$ | $const \leftarrow q.const \cup \{(b, \text{'}true)\}$ |
| $\times\,(q_1, q_2)$, $\overset{=}{\bowtie}_{b_1=b_2}\,(q_1, q_2)$, $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1, q_2)$ | $const \leftarrow q_1.const \cup q_2.const$ |
| $\delta\,(q)$ | $const \leftarrow q.const$ |
| $\dot\cup_{b_1,\ldots,b_n}\,(q_1, q_2)$ | $const \leftarrow \{(b_i, v) \mid (b_i, v) \in q_1.const, (b_i, v) \in q_2.const\}$ |
| $\setminus_{b_1,\ldots,b_n}\,(q_1, q_2)$ | $const \leftarrow \{(b_i, v) \mid (b_i, v) \in q_1.const\}$ |
| $\text{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}\,(q)$ | $const \leftarrow \{(b_{grp}, v) \mid (b_{grp}, v) \in q.const\} \cup \{(a_i, v) \mid (b_i, v) \in q.const, \circ_i \in \{\text{AVG}, \text{MAX}, \text{MIN}, \text{THE}\}\}$ |
| $\begin{array}{\|c\|}\hline a_1 \ldots a_n \\ \hline \varnothing \\ \hline\end{array}$ | $const \leftarrow \varnothing$ |
| $\begin{array}{\|c\|c\|c\|}\hline a_1 & \ldots & a_n \\ \hline & & \\ \hline\end{array}$, $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|c\|}{name} \\ \hline a_1 & \ldots & a_n \\ \hline\end{array}$ | $const \leftarrow \{(a_i, \text{CONST-VAL}(a_i)) \mid i \in \{1, \ldots, n\}, \text{CONST}(a_i)\}$ |
| $@_{a:val}\,(q)$ | $const \leftarrow q.const \cup \{(a, val)\}$ |
| $\textcircled{r}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $const \leftarrow q.const$ |
| $\textcircled{$\wedge$}_{a:\langle b_1,b_2\rangle}\,(q)$ | $const \leftarrow q.const \cup \{(a, v_1 \wedge v_2) \mid (b_1, v_1) \in q.const, (b_2, v_2) \in q.const\}$ |
| $\textcircled{$\vee$}_{a:\langle b_1,b_2\rangle}\,(q)$ | $const \leftarrow q.const \cup \{(a, v_1 \vee v_2) \mid (b_1, v_1) \in q.const, (b_2, v_2) \in q.const\}$ |
| $\textcircled{$\neg$}_{a:\langle b\rangle}\,(q)$ | $const \leftarrow q.const \cup \{(a, \neg v) \mid (b, v) \in q.const\}$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}\,(q)$, $\vec{\triangledown}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$, $\triangledown_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$, $\lrcorner^{\alpha,v}_{a:\langle b\rangle}\,(q)$, $\varphi_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}\,(q)$ | $const \leftarrow q.const$ |

Table 4.3: Bottom-up inference of the constant property *const* that derives the set of constant columns together with their corresponding value.

together with their corresponding constant values. New constant pairs are introduced for @ operators and constant columns in base tables. The constant information is propagated to the downstream plan operators.

The inference of *const*, for example, propagates constants in a union $\dot{\cup}$, if two matching columns share the same constant value, and maintains the constant information of some aggregates for grouping operators. For the Boolean operators $\owedge$, $\ovee$, and $\ominus$ the output values are calculated based on their constant input information. For the operators summarized by ⓡ, the result values are not precomputed, to avoid inconsistent changes: The rounding precision of operations on floating numbers, for example, might differ between the optimizer and the back-end database system.

Based on the constant property *const*, Rewrites 6 and 8 replace a selection operator. Note how the successful application of Rewrite 6 enables Rewrites 1 to 3 to further simplify the query plan.

$$\frac{(b, \text{`}\mathit{false}) \in \mathit{const} \qquad \{c_1, \ldots, c_n\} = \mathit{cols}}{\sigma_b\,(q) \rightarrow \begin{array}{|c|}\hline c_1 \;\ldots\; c_n \\ \hline \varnothing \\ \hline\end{array}}\,(6) \qquad \frac{\{(a, v), (b, v)\} \subseteq \mathit{const}}{q_1 \overset{=}{\bowtie}_{a=b}\, q_2 \rightarrow q_1 \times q_2}\,(7)$$

$$\frac{(b, \text{`}\mathit{true}) \in \mathit{const}}{\sigma_b\,(q) \rightarrow q}\,(8) \qquad \frac{\mathit{const} \neq \varnothing \qquad \{(c_1, v_1), \ldots, (c_n, v_n)\} = \mathit{const}}{\delta(q) \rightarrow \delta\left(\pi_{\mathit{cols} \setminus \mathit{const}}(q)\right) \times \begin{array}{|c|c|c|}\hline c_1 & \ldots & c_n \\ \hline v_1 & & v_n \\ \hline\end{array}}\,(9)$$

$$\frac{\{(b_1, v_1), \ldots, (b_n, v_n)\} \subseteq q.\mathit{const} \qquad \{\circ_1, \ldots, \circ_n\} \subseteq \{\text{AVG}, \text{MIN}, \text{MAX}, \text{THE}\}}{\text{GRP}_{a_1:\circ_1(b_1), \ldots, a_n:\circ_n(b_n)/b_{grp}}\,(q) \rightarrow \delta\left(\pi_{b_{grp}}(q)\right) \times \begin{array}{|c|c|c|}\hline a_1 & \ldots & a_n \\ \hline v_1 & & v_n \\ \hline\end{array}}\,(10)$$

The right-hand side of Rewrite 9 introduces a projection based on the argument *cols* $\setminus$ *const*, indicating the set of non-constant columns. In the remainder of this work, we assume that $\cap$, $\cup$, and $\setminus$ are able to correctly intersect, union, and subtract arguments based on their column names only: The argument *cols* $\setminus$ *const*, for example, matches a column $c_1 \in \mathit{cols}$ with a pair $(c_1, v_1) \in \mathit{const}$.

Rewrite 10 considers, in addition to the well-known aggregates AVG, MAX, and MIN, the aggregate THE available, for example, in Haskell [73]. THE returns an arbitrary element of a non-empty list of equal elements.

The *key* property inference (Tables 4.4 and 4.5) collects the sets of columns forming candidate keys. The bottom-up DAG traversal introduces single-column keys at the base tables based on the cardinality[2] and the existing key information (KEY($a$)). Operators $\times$, $\overset{=}{\bowtie}$, $\bowtie$, $\delta$, $\vec{\#}$, and ⊿ furthermore introduce keys consisting of multiple columns.

Based on the key property, Rewrites 11 and 12 throw away superfluous $\delta$ and GRP operators.

$$\frac{k \in q.\mathit{key}}{\delta(q) \rightarrow q}\,(11) \qquad \frac{\{b_{grp}\} \in q.\mathit{key} \qquad \{\circ_1, \ldots, \circ_n\} \not\subseteq \{\text{COUNT}\}}{\text{GRP}_{a_1:\circ_1(b_1), \ldots, a_n:\circ_n(b_n)/b_{grp}}\,(q) \rightarrow \pi_{b_{grp}, a_1:b_1, \ldots, a_n:b_n}(q)}\,(12)$$

[2]Table 4.6 shows the inference of *card1*. *card1* records if an operator produces exactly one row.

| **Key Property** *key* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}(q)$ | $key \leftarrow \{\{a_i \mid b_i \in k\} \mid k \in q.key, k \subseteq \{b_1,\dots,b_n\}\}$ |
| $\sigma_b(q)$ | $key \leftarrow q.key$ |
| $\times(q_1,q_2)$ | $key \leftarrow \{k \mid k \in q_1.key, q_2.card1\} \cup$<br>$\{k \mid k \in q_2.key, q_1.card1\} \cup$<br>$\{k_1 \cup k_2 \mid k_1 \in q_1.key, k_2 \in q_2.key\}$ |
| $\bar{\ltimes}_{b_1=b_2}(q_1,q_2)$ | $key \leftarrow \{k \mid k \in q_1.key, q_2.card1\} \cup$<br>$\{k \mid k \in q_2.key, q_1.card1\} \cup$<br>$\{k \mid k \in q_1.key, \{b_2\} \in q_2.key\} \cup$<br>$\{k \mid k \in q_2.key, \{b_1\} \in q_1.key\} \cup$<br>$\{(k_1 \setminus \{b_1\}) \cup k_2 \mid \{b_2\} \in q_2.key,$<br>$k_1 \in q_1.key,$<br>$k_2 \in q_2.key\} \cup$<br>$\{k_1 \cup (k_2 \setminus \{b_2\}) \mid \{b_1\} \in q_1.key,$<br>$k_1 \in q_1.key,$<br>$k_2 \in q_2.key\} \cup$<br>$\{k_1 \cup k_2 \mid k_1 \in q_1.key, k_2 \in q_2.key\}$ |
| $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1} \wedge \dots \wedge b_{1\cdot n}\theta_n b_{2\cdot n}}(q_1,q_2)$ | $key \leftarrow \{k \mid k \in q_1.key, q_2.card1\} \cup$<br>$\{k \mid k \in q_2.key, q_1.card1\} \cup$<br>$\{k \mid k \in q_1.key, i \in \{1,\dots,n\},$<br>$\theta_i \in \{=\}, \{b_{2\cdot i}\} \in q_2.key\} \cup$<br>$\{k \mid k \in q_2.key, i \in \{1,\dots,n\},$<br>$\theta_i \in \{=\}, \{b_{1\cdot i}\} \in q_1.key\} \cup$<br>$\{k_1 \cup k_2 \mid k_1 \in q_1.key, k_2 \in q_2.key\}$ |
| $\delta(q)$ | $key \leftarrow q.key \cup \{cols\}$ |
| $\dot{\cup}_{b_1,\dots,b_n}(q_1,q_2)$ | $key \leftarrow \{\{b_i\} \mid \{b_i\} \in q_1.key, \{b_i\} \in q_2.key,$<br>$(b_i : d_1) \in q_1.dom,$<br>$(b_i : d_2) \in q_2.dom,$<br>$disjointDom(d_1,d_2)\}$ |
| $\setminus_{b_1,\dots,b_n}(q_1,q_2)$ | $key \leftarrow \{k \mid k \in q_1.key, k \subseteq \{b_1,\dots,b_n\}\}$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\dots,a_n:\circ_n(b_n)/b_{grp}}(q)$ | $key \leftarrow \{\{b_{grp}\}\}$ |
| $\begin{array}{c} a_1 \dots a_n \\ \hline \varnothing \end{array}$, $\begin{array}{ccc} \multicolumn{3}{c}{a_1 \dots a_n} \\ \hline \; & \; & \; \end{array}$, $\begin{array}{ccc} \multicolumn{3}{c}{name} \\ \hline a_1 & \dots & a_n \end{array}$ | $key \leftarrow \{\{a\} \mid a \in \{a_1,\dots,a_n\}, card1 \vee \mathrm{KEY}(a)\}$ |

Table 4.4: Bottom-up inference of the key property *key* that derives the sets of key candidate columns.

| **Key Property** *key* **(continued)** | |
|---|---|
| $ⓡ_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $key \leftarrow q.key \cup \{\{a\} \mid q.card1\}$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}\,(q)$ | $key \leftarrow q.key \cup \{\{a\} \mid q.card1\} \cup \{\{a, b_{grp}\}\}$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $key \leftarrow q.key \cup \{\{a\}\}$ |
| $\vec{⛉}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$,<br>$⛉_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $key \leftarrow q.key \cup \{\{a\} \mid q.card1\} \cup$<br>$\{\{a\} \cup (k \setminus \{b_1,\ldots,b_n\}) \mid k \in q.key,$<br>$k \cap \{b_1,\ldots,b_n\} \neq \varnothing\}$ |
| $⌙^{\alpha,v}_{a:\langle b\rangle}\,(q)$ | $key \leftarrow \{\{a\} \mid q.card1\} \cup$<br>$\{\{a\} \cup k \mid k \in q.key\} \cup$<br>$\{\{a\} \cup (k \setminus \{b\}) \mid k \in q.key, k \cap \{b\} \neq \varnothing,$<br>$\alpha \in \{\texttt{attribute}, \texttt{child},$<br>$\texttt{self}\}\} \cup$<br>$\{k \mid k \in q.key, \alpha \in \{\texttt{self}\}\}$ |
| $♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}\,(q)$ | $key \leftarrow q.key$ |

Table 4.5: Bottom-up inference of the key property *key* that derives the sets of key candidate columns.

| **Single-Row Property** *card1* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}\,(q)$ | $card1 \leftarrow q.card1$ |
| $\sigma_b\,(q)$ | $card1 \leftarrow$ ‘$false$ |
| $\times\,(q_1, q_2)$ | $card1 \leftarrow q_1.card1 \wedge q_2.card1$ |
| $\overset{=}{\bowtie}_{b_1=b_2}\,(q_1, q_2)$, $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1, q_2)$ | $card1 \leftarrow$ ‘$false$ |
| $\delta\,(q)$ | $card1 \leftarrow q.card1$ |
| $\dot\cup_{b_1,\ldots,b_n}\,(q_1, q_2)$, $\setminus_{b_1,\ldots,b_n}\,(q_1, q_2)$,<br>$\mathrm{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}\,(q)$ | $card1 \leftarrow$ ‘$false$ |
| $\begin{array}{\|c\|c\|c\|}\hline a_1 & \ldots & a_n \\ \hline & & \\ \hline\end{array}$ | $card1 \leftarrow \mathrm{CARD} = 1$ |
| $\begin{array}{\|c\|}\hline a_1\ \ldots\ a_n \\ \hline \varnothing \\ \hline\end{array}$, $\overset{name}{\begin{array}{\|c\|c\|c\|}\hline a_1 & \ldots & a_n \\ \hline\end{array}}$ | $card1 \leftarrow$ ‘$false$ |
| $ⓡ_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$, $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}\,(q)$,<br>$\vec{⛉}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$, $⛉_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $card1 \leftarrow q.card1$ |
| $⌙^{\alpha,v}_{a:\langle b\rangle}\,(q)$ | $card1 \leftarrow$ ‘$false$ |
| $♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}\,(q)$ | $card1 \leftarrow q.card1$ |

Table 4.6: Bottom-up inference of the single-row property *card1* that derives whether the evaluation of an operator produces exactly one row at runtime.

The more elaborate *key* inference rules for equi-joins ($\bar{\bowtie}$, $\bowtie$ with $\theta_i \in \{=\}$) and rank operators ($\varrho$) follow from functional dependencies. [113, Section 5.2.1] gives a more detailed account of the equi-join *key* inference. The $\dot\cup$ operator propagates key columns, whenever its input columns store disjoint values.[3] The row numbering operator $\vec{\#}$ provides different keys, based on the existence of its grouping criterion $b_{grp}$: Either the newly generated column $a$ on its own or the combination of the columns $\{a, b_{grp}\}$ form a new key. The key property inference of ⍁ takes into account XML specific knowledge about the semantics of axis $\alpha$.

The single-row property *card1* (Table 4.6) is the second property—in addition to property *empty*—that enables decisions based on the cardinality of literal tables and their downstream plans. The single-row property is extensively used in the key property inference and plays an important role in the rewrites described in Section 4.3. *card1* cannot be guaranteed for $\dot\cup$ and operators performing variants of selections ($\sigma$,$\bar{\bowtie}$,$\bowtie$, and ⍁). For most other operators however the cardinality stays the same and *card1* is propagated.

For the query plan of Query $Q_3$ (Figure 2.9; page 23) the single-row property is used to infer the keyness of column res in operator ⍁. The key information is then propagated to the downstream operators where it enables the application of Rewrite 11 at operator $\delta$.

The domain property *dom* (Table 4.7) creates abstract active domains and establishes a relationship between them for all columns. Two abstract domains $d_1$ and $d_2$ can be compared for their subdomain relationship or domain disjointness (*subDom* $(d_1, d_2)$ or *disjointDom* $(d_1, d_2)$). All domains are a subdomain of the domain universe $\mathcal{U}$. An entry in *dom* either consists of the relationship $d_1 \overset{\subseteq}{\leftarrow} d_2$ between the right-hand side (super-)domain $d_2$ and a domain identifier $d_1$ or the disjointness relationship $d_1 \,||\, d_2$. $\mathcal{D}(c)$ describes a domain identifier taking column $c$ and the current operator into account.

Because of the abstract nature of the domain property—the property is independent of any values—only columns generated by the same operator may be in a subdomain or disjoint domain relationship. In the scope of loop lifted algebra plans with operator sharing, this abstract domain property still plays an import role.

The bottom-up domain property inference shown in Table 4.7 introduces new subdomain relationships for the operators, potentially filtering out some rows. The inference introduces new domains—being direct subdomains of $\mathcal{U}$—for base tables and operators that introduce new columns and preserves the input domains, if the value distribution does not change.

For an equi-join operator $\bar{\bowtie}$ the domain property inference aligns the domain of the two join columns. A distinct operator $\delta$ removes duplicates only and thus does not change the domains. The union operator $\dot\cup$ builds new subdomains based

[3]Peek forward to Table 4.7 for details on the domain property *dom* inference.

| **Domain Property** *dom* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}\,(q)$ | $dom \leftarrow \{(a_i, d) \mid (b_i, d) \in q.dom\}$ |
| $\sigma_b\,(q)$ | $dom \leftarrow \{(c, \mathcal{D}(c) \overset{\subseteq}{\leftarrow} d) \mid (c, d) \in q.dom\}$ |
| $\times\,(q_1, q_2)$ | $dom \leftarrow \{(c, \mathcal{D}(c) \overset{\subseteq}{\leftarrow} d) \mid (c, d) \in q_1.dom\} \cup$<br>$\{(c, \mathcal{D}(c) \overset{\subseteq}{\leftarrow} d) \mid (c, d) \in q_2.dom\}$ |
| $\overset{=}{\bowtie}_{b_1=b_2}\,(q_1, q_2)$ | $dom \leftarrow \{(b_1, \mathcal{D}(b_1) \overset{\subseteq}{\leftarrow} \cup(d_1, d_2)),$<br>$(b_2, \mathcal{D}(b_1) \overset{\subseteq}{\leftarrow} \cup(d_1, d_2))$<br>$\mid (b_1, d_1) \in q_1.dom,\ (b_2, d_2) \in q_2.dom\} \cup$<br>$\{(c, \mathcal{D}(c) \overset{\subseteq}{\leftarrow} d) \mid (c, d) \in q_1.dom, c \neq b_1\} \cup$<br>$\{(c, \mathcal{D}(c) \overset{\subseteq}{\leftarrow} d) \mid (c, d) \in q_2.dom, c \neq b_2\}$ |
| $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\dots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1, q_2)$ | $dom \leftarrow \{(c, \mathcal{D}(c) \overset{\subseteq}{\leftarrow} d) \mid (c, d) \in q_1.dom\} \cup$<br>$\{(c, \mathcal{D}(c) \overset{\subseteq}{\leftarrow} d) \mid (c, d) \in q_2.dom\}$ |
| $\delta\,(q)$ | $dom \leftarrow q.dom$ |
| $\dot\cup_{b_1,\dots,b_n}\,(q_1, q_2)$ | $dom \leftarrow \{(b_i, \mathcal{D}(b_i) \overset{\subseteq}{\leftarrow} lca(d_1, d_2))$<br>$\mid (b_i, d_1) \in q_1.dom, (b_i, d_2) \in q_2.dom\}$ |
| $\setminus_{b_1,\dots,b_n}\,(q_1, q_2)$ | $dom \leftarrow \{(b_i, \mathcal{D}(b_i) \overset{\subseteq}{\leftarrow} d) \mid (b_i, d) \in q_1.dom\}$ |
| $\setminus_{b_1}\,(q_1, q_2)$ | $dom \leftarrow \{(b_1, \mathcal{D}(b_1) \overset{\subseteq}{\leftarrow} d) \mid (b_1, d) \in q_1.dom\} \cup$<br>$\{(b_1, \mathcal{D}(b_1) \mid\mid d) \mid (b_1, d) \in q_2.dom\}$ |
| $\text{GRP}_{a_1:\circ_1(b_1),\dots,a_n:\circ_n(b_n)/b_{grp}}\,(q)$ | $dom \leftarrow \{(b_{grp}, d) \mid (b_{grp}, d) \in q.dom\} \cup$<br>$\{(a_i, \mathcal{D}(a_i) \overset{\subseteq}{\leftarrow} d)$<br>$\mid (b_i, d) \in q.dom,$<br>$\circ_i \in \{\text{MAX}, \text{MIN}, \text{THE}\}\} \cup$<br>$\{(a_i, \mathcal{D}(a_i) \overset{\subseteq}{\leftarrow} \mathcal{U})$<br>$\mid i \in \{1, \dots, n\},$<br>$\circ_i \notin \{\text{MAX}, \text{MIN}, \text{THE}\}\}$ |
| $\begin{array}{\|c\|}\hline a_1 \dots a_n \\ \hline \varnothing \\ \hline\end{array}$, $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|c\|}{a_1 \dots a_n} \\ \hline \ & \ & \ \\ \hline\end{array}$, $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|l}{name} \\ \hline a_1 & \dots & a_n \\ \hline\end{array}$ | $dom \leftarrow \{(a_i, \mathcal{D}(a_i) \overset{\subseteq}{\leftarrow} \mathcal{U}) \mid i \in \{1, \dots, n\}\}$ |
| $\textcircled{r}_{a:\langle b_1,\dots,b_n\rangle}\,(q)$,<br>$\vec{\#}_{a:\langle b_1,\dots,b_n\rangle/b_{grp}}\,(q)$,<br>$\vec{\boxed{\gg}}_{a:\langle b_1,\dots,b_n\rangle}\,(q)$, $\boxed{\gg}_{a:\langle b_1,\dots,b_n\rangle}\,(q)$ | $dom \leftarrow q.dom\ \cup\ \{(a : \mathcal{D}(a) \overset{\subseteq}{\leftarrow} \mathcal{U})\}$ |
| $⎍^{\alpha,v}_{a:\langle b\rangle}\,(q)$ | $dom \leftarrow \{(c, \mathcal{D}(c) \overset{\subseteq}{\leftarrow} d) \mid (c, d) \in q.dom\} \cup$<br>$\{(a : \mathcal{D}(a) \overset{\subseteq}{\leftarrow} \mathcal{U})\}$ |
| $♀_{b_1,b_2,\langle b_3,\dots,b_n\rangle}\,(q)$ | $dom \leftarrow q.dom$ |

Table 4.7: Bottom-up inference of the domain property *dom* that derives the domain relationship of its columns.

on *lca* ()—the lowest common ancestor domain of the input domains. Disjoint domains are introduced only for difference operators operating on a single column $q_1 \setminus_b q_2$ as we can guarantee that the resulting domain is disjoint of the domain in $q_2$.

If property *dom* infers that the active domain on the left-hand side of an one-column difference operator is already a subdomain of the right-hand side, Rewrite 13 can replace the difference by an empty table:

$$\frac{(b, d_1) \in q_1.dom \qquad (b, d_2) \in q_2.dom \qquad subDom(d_1, d_2)}{q_1 \setminus_b q_2 \rightarrow \boxed{\begin{array}{c} b \\ \hline \emptyset \end{array}}} \quad (13)$$

**Interaction between the key and domain properties.** Most equi-join operators $q_1 \overset{=}{\bowtie}_{b_1=b_2} q_2$ introduced in the loop lifted compilation scheme compare columns that originate in the same (numbering) operator. Based on the key information $\{b_2\} \in q_2.key$, we know that an equi-join will match every row in input $q_1$ with at most one row in $q_2$. For every possible value of $b_1$ in $q_1$, we can furthermore derive, based on the domain information $(b_1, d_1) \in q_1.dom \wedge (b_2, d_2) \in q_2.dom \wedge subDom\,(d_1, d_2)$, that $q_2$ provides at least one matching value in column $b_2$. The combination of key and domain property information ensures that every row in $q_1$ finds *exactly one* match in $q_2$. This observation is the most important aspect of the domain property inference, as it paves the ways for many important rewrites in the following sections.

The functional dependency property *fd* (Table 4.8) performs a bottom-up traversal of the DAG to collect a set of functional dependencies $(a \rightarrow b)$, indicating that column $b$ functionally depends on column $a$. The property relies heavily on the key property to introduce new functional dependencies. *fd* is especially useful to describe the relationship between the input and result columns of the ranking operators $\vec{⩔}$ and ⩔, and operator ⎿.

Rewrite 14 uses the functional dependency property to turn a grouping operator into a distinct operator, if all aggregate columns $b_1, \ldots, b_n$ functionally depend on the grouping column $b_{grp}$ (and the aggregates do not depend on the input cardinality):

$$\frac{\{(b_{grp} \rightarrow b_1), \ldots, (b_{grp} \rightarrow b_n)\} \subseteq q.fd \qquad \{\circ_1, \ldots, \circ_n\} \subseteq \{\text{MIN}, \text{MAX}, \text{AVG}, \text{THE}\}}{\text{GRP}_{a_1:\circ_1(b_1), \ldots, a_n:\circ_n(b_n)/b_{grp}}\,(q) \rightarrow \delta\left(\pi_{b_{grp}, a_1:b_1, \ldots, a_n:b_n}(q)\right)} \quad (14)$$

## 4.2.3 An Alternative to Selection Pushdown

Selection pushdown as described in database textbooks [38,112] takes an arbitrarily complex selection, merges parts of it with other selections, cross products, or joins, and pushes the remaining predicates down through unaffected projections, cross products, and joins.

| **Functional Dependency Property** *fd* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}\,(q)$ | $fd \leftarrow \{(a_i \rightarrow a_j) \mid (b_i \rightarrow b_j) \in q.fd\}$ |
| $\sigma_b\,(q)$ | $fd \leftarrow q.fd$ |
| $\times\,(q_1, q_2)$ | $fd \leftarrow q_1.fd \,\cup\, q_2.fd \,\cup$<br>$\{(k \rightarrow c) \mid \{k\} \in key, c \in cols, k \neq c\}$ |
| $\bar{\bowtie}_{b_1=b_2}\,(q_1, q_2)$ | $fd \leftarrow q_1.fd \,\cup\, q_2.fd \,\cup$<br>$\{(k \rightarrow c) \mid \{k\} \in key, c \in cols, k \neq c\} \cup$<br>$\{(b_2 \rightarrow c_1) \mid (b_1 \rightarrow c_1) \in q_1.fd\} \cup$<br>$\{(b_1 \rightarrow c_2) \mid (b_2 \rightarrow c_2) \in q_2.fd\} \cup$<br>$\{(k \rightarrow c) \mid \{b_1\} \in q_1.key,$<br>$\{k\} \in q_1.key, k \neq b_1,$<br>$(b_2 \rightarrow c) \in q_2.fd\} \cup$<br>$\{(k \rightarrow c) \mid \{b_2\} \in q_2.key,$<br>$\{k\} \in q_2.key, k \neq b_2,$<br>$(b_1 \rightarrow c) \in q_1.fd\}$ |
| $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1} \wedge \cdots \wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1, q_2)$ | $fd \leftarrow q_1.fd \,\cup\, q_2.fd \,\cup$<br>$\{(k \rightarrow c) \mid \{k\} \in key, c \in cols, k \neq c\}$ |
| $\delta\,(q)$ | $fd \leftarrow q.fd$ |
| $\dot\cup_{b_1,\ldots,b_n}\,(q_1, q_2)$ | $fd \leftarrow \{(k \rightarrow b_i) \mid \{k\} \in key, k \neq b_i\} \cup$<br>$\{(b_i \rightarrow o) \mid (o, v) \in const, o \neq b_i\}$ |
| $\setminus_{b_1,\ldots,b_n}\,(q_1, q_2)$ | $fd \leftarrow \{(b_i \rightarrow b_j) \mid (b_i \rightarrow b_j) \in q_1.fd\}$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}\,(q)$ | $fd \leftarrow \{(b_{grp} \rightarrow a_i) \mid i \in \{1, \ldots, n\}\}$ |
| $\begin{array}{\|c\|}\hline a_1 \ldots a_n \\ \hline \varnothing \\ \hline\end{array}$, $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|c\|}{a_1 \ldots a_n} \\ \hline \; & \; & \; \\ \hline\end{array}$, $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|l\|}{name} \\ \hline a_1 & \ldots & a_n \\ \hline\end{array}$ | $fd \leftarrow \{(k \rightarrow c) \mid \{k\} \in key, c \in cols, k \neq c\} \cup$<br>$\{(c \rightarrow o) \mid (o, v) \in const, c \in cols, o \neq c\}$ |
| $@_{a:val}\,(q)$ | $fd \leftarrow \{(c \rightarrow a) \mid c \in q.cols\}$ |
| $\textcircled{r}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $fd \leftarrow q.fd \,\cup$<br>$\{(k \rightarrow c) \mid \{k\} \in key, c \in cols, k \neq c\} \cup$<br>$\{(c \rightarrow a) \mid (c \rightarrow b_1) \in q.fd, \ldots,$<br>$(c \rightarrow b_n) \in q.fd\}$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}\,(q)$ | $fd \leftarrow q.fd$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $fd \leftarrow q.fd \,\cup\, \{(a \rightarrow c) \mid c \in q.cols\}$ |
| $\vec{⛉}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$, $⛉_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $fd \leftarrow q.fd \,\cup\, \{(a \rightarrow b_i) \mid i \in \{1, \ldots, n\}\}$ |
| $⎍^{\alpha,v}_{a:\langle b\rangle}\,(q)$ | $fd \leftarrow q.fd \,\cup$<br>$\{(k \rightarrow c) \mid \{k\} \in key, c \in cols, k \neq c\} \,\cup$<br>$\{(a \rightarrow b) \mid \alpha \in \{\texttt{attribute}, \texttt{child}, \texttt{self}\}\}$ |
| $♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}\,(q)$ | $fd \leftarrow q.fd$ |

Table 4.8: Bottom-up inference of the functional dependency property *fd* that derives the set of functional dependencies.

| **Reference Counting Property** *#ref* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}(q)$, $\sigma_b(q)$ | $q.\#\mathit{ref} \leftarrow q.\#\mathit{ref}+1$ |
| $\times(q_1,q_2)$, $\overset{\bar{=}}{\bowtie}_{b_1=b_2}(q_1,q_2)$, $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}(q_1,q_2)$ | $q_1.\#\mathit{ref} \leftarrow q_1.\#\mathit{ref}+1$ |
| $\times(q_1,q_2)$, $\overset{\bar{=}}{\bowtie}_{b_1=b_2}(q_1,q_2)$, $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}(q_1,q_2)$ | $q_2.\#\mathit{ref} \leftarrow q_2.\#\mathit{ref}+1$ |
| $\delta(q)$ | $q.\#\mathit{ref} \leftarrow q.\#\mathit{ref}+1$ |
| $\dot{\cup}_{b_1,\ldots,b_n}(q_1,q_2)$, $\setminus_{b_1,\ldots,b_n}(q_1,q_2)$ | $q_1.\#\mathit{ref} \leftarrow q_1.\#\mathit{ref}+1$ |
| $\dot{\cup}_{b_1,\ldots,b_n}(q_1,q_2)$, $\setminus_{b_1,\ldots,b_n}(q_1,q_2)$ | $q_2.\#\mathit{ref} \leftarrow q_2.\#\mathit{ref}+1$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}(q)$, $\textcircled{r}_{a:\langle b_1,\ldots,b_n\rangle}(q)$, $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(q)$, $\vec{\veebar}_{a:\langle b_1,\ldots,b_n\rangle}(q)$, $\veebar_{a:\langle b_1,\ldots,b_n\rangle}(q)$, $\lrcorner^{\alpha,v}_{a:\langle b\rangle}(q)$, $\varphi_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}(q)$ | $q.\#\mathit{ref} \leftarrow q.\#\mathit{ref}+1$ |

Table 4.9: Top-down inference of the reference counting property *#ref* that derives the number of parent operators. (*#ref* is seeded with 0.)

In our algebra, a selection with a single predicate consists of at least two operators: a selection $\sigma$ and a comparison (e.g., $\circleddash$). Although this design decision prohibits a standard approach to selection pushdown, it fits our compositional translation in which selections and comparisons are generated by separate compilation rules. This separation of concerns furthermore simplifies the optimization process. Property inference and rewrite rules need to cope only with simple operators—as opposed to operators whose arguments themselves consist of complex expression trees. We support an alternative variant of selection pushdown by means of a property inference phase that traces selected columns together with their expected Boolean value (see property *req* below) in a top-down DAG traversal.

A top-down property inference phase seeds the property value for every operator. A second run then combines, for every operator $q$, the inferred property information of its parent operators $(q_{p_1},\ldots,q_{p_n})$. The traversal of $q$'s children thus proceeds only after all parents $q_{p_1},\ldots,q_{p_n}$ pushed down their properties to $q$.

The reference counting property *#ref* (Table 4.9) collects the number of parent operators for each operator $q$, based on which all other top-down inferred properties can easily detect, when an operator is visited the last time during a top-down property inference. For all operators *#ref* is seeded with 0 and—as opposed to the other top-down properties—traverses the children of an operator the first time the operator is reached.

The required value property *req* (Table 4.10) collects the selection columns and

| **Required Value Property** *req* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}\,(q)$ | $q.req\ \leftarrow \{(b_i, v)\,\vert\,(a_i : v) \in req\}$ |
| $\sigma_b\,(q)$ | $q.req\ \leftarrow \{(b, \text{‘}true)\} \cup \{(c, v)\,\vert\,(c, v) \in req, c \neq b\}$ |
| $\times\,(q_1, q_2),\ \bar{\ltimes}_{b_1=b_2}\,(q_1, q_2),$ $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1, q_2)$ | $q_1.req \leftarrow \{(c, v)\,\vert\,(c, v) \in req, c \in q_1.cols\}$ |
| $\times\,(q_1, q_2),\ \bar{\ltimes}_{b_1=b_2}\,(q_1, q_2),$ $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1, q_2)$ | $q_2.req \leftarrow \{(c, v)\,\vert\,(c, v) \in req, c \in q_2.cols\}$ |
| $\delta\,(q)$ | $q.req\ \leftarrow req$ |
| $\dot{\cup}_{b_1,\ldots,b_n}\,(q_1, q_2)$ | $q_1.req \leftarrow \{(b_i, v)\,\vert\,(b_i, v) \in req\}$ |
| $\dot{\cup}_{b_1,\ldots,b_n}\,(q_1, q_2)$ | $q_2.req \leftarrow \{(b_i, v)\,\vert\,(b_i, v) \in req\}$ |
| $\setminus_{b_1,\ldots,b_n}\,(q_1, q_2)$ | $q_1.req \leftarrow \{(b_i, v)\,\vert\,(b_i, v) \in req\}$ |
| $\setminus_{b_1,\ldots,b_n}\,(q_1, q_2)$ | $q_2.req \leftarrow \emptyset$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}\,(q)$ | $q.req\ \leftarrow \emptyset$ |
| $\textcircled{r}_{\mathsf{a}:\langle\mathsf{b}_1,\ldots,\mathsf{b}_\mathsf{n}\rangle}\,(q)$ | $q.req\ \leftarrow \{(c, v)\,\vert\,(c, v) \in req, c \neq a\}$ |
| $\textcircled{\wedge}_{a:\langle b_1,b_2\rangle}\,(q)$ | $q.req\ \leftarrow \{(c, v)\,\vert\,(c, v) \in req, c \neq a\} \cup$<br>$\{(b_1, \text{‘}true), (b_2, \text{‘}true)\,\vert\,(a, \text{‘}true) \in req\}$ |
| $\textcircled{\vee}_{a:\langle b_1,b_2\rangle}\,(q)$ | $q.req\ \leftarrow \{(c, v)\,\vert\,(c, v) \in req, c \neq a\} \cup$<br>$\{(b_1, \text{‘}false), (b_2, \text{‘}false)\,\vert\,(a, \text{‘}false) \in req\}$ |
| $\textcircled{\neg}_{a:\langle b\rangle}\,(q)$ | $q.req\ \leftarrow \{(c, v)\,\vert\,(c, v) \in req, c \neq a\} \cup$<br>$\{(b, \neg v)\,\vert\,(a, v) \in req\}$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}\,(q),$ $\vec{\boxed{\gg}}_{a:\langle b_1,\ldots,b_n\rangle}\,(q)$ | $q.req\ \leftarrow \emptyset$ |
| $\boxed{\gg}_{a:\langle b_1,\ldots,b_n\rangle}\,(q),\ \sqcup^{\alpha,v}_{a:\langle b\rangle}\,(q)$ | $q.req\ \leftarrow \{(c, v)\,\vert\,(c, v) \in req, c \neq a\}$ |
| $\varrho_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}\,(q)$ | $q.req\ \leftarrow \emptyset$ |

Table 4.10: Top-down inference of the required value property *req* that infers selection columns and their expected Boolean value. ($q.req$ is inferred only for $q.\#ref = 1$. *req* is seeded with $\emptyset$.)

their expected Boolean value and pushes this information down into the subplans. To avoid inconsistent requirements, the property limits, based on the reference counting property, the propagation of *req* to the operators with a single parent reference.[4]

For each selection operator $\sigma$, the inference of *req* introduces new entry consisting of the column and its corresponding expected Boolean value. The operators $\owedge$, $\ovee$, and $\ominus$ propagate the information to additional columns. Operators $\setminus$, GRP, $\vec{\#}$, and $\vec{\veebar}$ reset the required value property, to prohibit rewrites that may incorrectly change the cardinality of an input. All other operators propagate the set of required value pairs to their children.

Rewrites 15 and 16 use property *req* to detect selection criteria that would reject all incoming rows. In consequence, the operators are replaced by empty literal tables—which, in turn, triggers Rewrites 1 to 3.

$$\frac{(b, \text{`}\mathit{false}) \in \mathit{req} \qquad \{c_1, \dots, c_n\} = \mathit{cols}}{\sigma_b\,(q) \rightarrow \begin{array}{|c|}\hline c_1 \,\dots\, c_n \\ \hline \varnothing \\ \hline \end{array}} \tag{15}$$

$$\frac{(a, \neg \mathit{val}) \in \mathit{req} \qquad \{c_1, \dots, c_n\} = \mathit{cols}}{@_{a:\mathit{val}}\,(q) \rightarrow \begin{array}{|c|}\hline c_1 \,\dots\, c_n \\ \hline \varnothing \\ \hline \end{array}} \tag{16}$$

$$\frac{(a, \text{`}\mathit{true}) \in \mathit{req}}{\owedge_{a:\langle b_1, b_2\rangle}\,(q) \rightarrow @_{a:\text{`}\mathit{true}}(\sigma_{b_1}(\sigma_{b_2}(q)))} \tag{17}$$

Although this form of selection pushdown seems rather limited, it is an important part of the house cleaning heuristic. Based on these rewrites, for example, XQuery's ubiquitous *existential semantics* are simplified in Pathfinder. A textbook-style selection pushdown process is orthogonal to this approach and may be additionally performed by the back-end.

## 4.2.4 An Alternative to Projection Pushdown

The required columns property *icols* (Table 4.11) and the accompanying rewrite rules (Rewrites 18 to 28) have an effect similar to a standard technique in relational query processors: *projection pushdown* [72]. In a top-down DAG traversal, the required columns property (*icols*) infers those columns that are strictly required to evaluate an operator: The compositional translation may have introduced operators that yield columns that remain unreferenced in the downstream plan. In a second traversal, Rewrites 18 to 28 prune unreferenced columns and remove any unnecessary operator.

The required column property is seeded with the empty set $\emptyset$. Any operator inherits the required columns of its parents and extends the set with all columns that are necessary to compute the operator's result. For the operators GRP, $\textcircled{\scriptsize T}$, $\vec{\#}$,

[4]Pathfinder uses an advanced variant of the required value property that additionally copes with the correct combination of (possibly conflicting) inferred properties.

| **Required Columns Property** *icols* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}\,(q)$ | $q.icols \leftarrow q.icols \cup \{b_i \mid a_i \in (icols \cap \{a_1,\dots,a_n\})\}$ |
| $\sigma_b\,(q)$ | $q.icols \leftarrow q.icols \cup icols \cup \{b\}$ |
| $\times\,(q_1,q_2)$ | $q_1.icols \leftarrow q_1.icols \cup (icols \cap q_1.cols)$ |
| $\times\,(q_1,q_2)$ | $q_2.icols \leftarrow q_2.icols \cup (icols \cap q_2.cols)$ |
| $\overset{=}{\bowtie}_{b_1=b_2}\,(q_1,q_2)$ | $q_1.icols \leftarrow q_1.icols \cup (icols \cap q_1.cols) \cup \{b_1\}$ |
| $\overset{=}{\bowtie}_{b_1=b_2}\,(q_1,q_2)$ | $q_2.icols \leftarrow q_2.icols \cup (icols \cap q_2.cols) \cup \{b_2\}$ |
| $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1,q_2)$ | $q_1.icols \leftarrow q_1.icols \cup (icols \cap q_1.cols) \cup \{b_{1\cdot1},\dots,b_{1\cdot n}\}$ |
| $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}\,(q_1,q_2)$ | $q_2.icols \leftarrow q_2.icols \cup (icols \cap q_2.cols) \cup \{b_{2\cdot1},\dots,b_{2\cdot n}\}$ |
| $\delta\,(q)$ | $q.icols \leftarrow q.icols \cup q.cols$ |
| $\dot\cup_{b_1,\dots,b_n}\,(q_1,q_2)$ | $q_1.icols \leftarrow q_1.icols \cup icols$ |
| $\dot\cup_{b_1,\dots,b_n}\,(q_1,q_2)$ | $q_2.icols \leftarrow q_2.icols \cup icols$ |
| $\setminus_{b_1,\dots,b_n}\,(q_1,q_2)$ | $q_1.icols \leftarrow q_1.icols \cup \{b_1,\dots,b_n\}$ |
| $\setminus_{b_1,\dots,b_n}\,(q_1,q_2)$ | $q_2.icols \leftarrow q_2.icols \cup \{b_1,\dots,b_n\}$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\dots,a_n:\circ_n(b_n)/b_{grp}}\,(q)$ | $q.icols \leftarrow q.icols \cup \{b_i \mid a_i \in (icols \cap \{a_1,\dots,a_n\})\} \cup \{b_{grp}\}$ |
| $Ⓡ_{a:\langle b_1,\dots,b_n\rangle}\,(q)$ | $q.icols \leftarrow q.icols \cup (icols \setminus \{a\}) \cup \{b_1,\dots,b_n \mid \{a\} \cap icols = \{a\}\}$ |
| $\vec{\#}_{a:\langle b_1,\dots,b_n\rangle/b_{grp}}\,(q)$ | $q.icols \leftarrow q.icols \cup (icols \setminus \{a\}) \cup \{b_1,\dots,b_n \mid \{a\} \cap icols = \{a\}\} \cup \{b_{grp} \mid \{a\} \cap icols = \{a\}\}$ |
| $\vec{\text{⛉}}_{a:\langle b_1,\dots,b_n\rangle}\,(q)$, $\text{⛉}_{a:\langle b_1,\dots,b_n\rangle}\,(q)$ | $q.icols \leftarrow q.icols \cup (icols \setminus \{a\}) \cup \{b_1,\dots,b_n \mid \{a\} \cap icols = \{a\}\}$ |
| $\text{⎍}^{\alpha,v}_{a:\langle b\rangle}\,(q)$ | $q.icols \leftarrow q.icols \cup (icols \setminus \{a\}) \cup \{b\}$ |
| $\text{♀}_{b_1,b_2,\langle b_3,\dots,b_n\rangle}\,(q)$ | $q.icols \leftarrow q.icols \cup \{b_1,\dots,b_n\}$ |

Table 4.11: Top-down inference of the required column property *icols* that infers the set of required input columns. (*icols* is seeded with $\emptyset$.)

$\vec{\curlyvee}$, and $\curlyvee$ the propagation of the *icols* property information depends on the use of their result column $a$: The argument columns are marked only if $a$ is a required input column ($\{a\} \cap icols = \{a\}$).

Rewrites 18 to 28 are, in contrast to the previous rewrite rules, interconnected. A rewrite results in a correct query plan only if all applicable *icols*-specific rewrites in the downstream plan have been performed first. Otherwise, a rewrite might lead to a plan where parent operators refer to already pruned columns.

$$\frac{icols \neq \varnothing \qquad \{a_1,\dots,a_n\} \setminus icols \neq \varnothing}{\pi_{a_1:b_1,\dots,a_n:b_n}(q) \rightarrow \pi_{\{a_1:b_1,\dots,a_n:b_n\} \cap icols}(q)} \quad (18) \qquad \frac{q_2.card1 \qquad q_2.icols = \varnothing}{q_1 \times q_2 \rightarrow q_1} \quad (19)$$

$$\frac{\begin{array}{c}\{b\} = q_2.icols \\ (b, d_1) \in q_1.dom \qquad (b, d_2) \in q_2.dom \\ subDom(d_1, d_2) \qquad \{b\} \in q_2.key\end{array}}{q_1 \overset{=}{\bowtie}_b q_2 \rightarrow q_1} \quad (20) \qquad \frac{\begin{array}{c}\{c_1,\dots,c_n\} = cols \setminus icols \\ \{a_1,\dots,a_n\} \subseteq icols \\ \{(a_1 \rightarrow c_1),\dots,(a_n \rightarrow c_n)\} \subseteq fd\end{array}}{\delta(q) \rightarrow \delta\left(\pi_{icols}(q)\right)} \quad (21)$$

$$\frac{const \setminus icols \neq \varnothing}{\delta(q) \rightarrow \delta\left(\pi_{cols \setminus (const \setminus icols)}(q)\right)} \quad (22) \qquad \frac{icols \neq \varnothing \qquad \{b_1,\dots,b_n\} \setminus icols \neq \varnothing}{q_1 \mathbin{\dot\cup}_{b_1,\dots,b_n} q_2 \rightarrow q_1 \mathbin{\dot\cup}_{\{b_1,\dots,b_n\} \cap icols} q_2} \quad (23)$$

$$\frac{\{a_1,\dots,a_n\} \cap icols = \varnothing}{\mathrm{GRP}_{a_1:\circ_1(b_1),\dots,a_n:\circ_n(b_n)/b_{grp}}(q) \rightarrow \delta(\pi_{b_{grp}}(q))} \quad (24)$$

$$\frac{icols \neq \varnothing \qquad \{a_1,\dots,a_n\} \setminus icols \neq \varnothing}{\mathrm{GRP}_{a_1:\circ_1(b_1),\dots,a_n:\circ_n(b_n)/b_{grp}}(q) \rightarrow \mathrm{GRP}_{\{a_1:\circ_1(b_1),\dots,a_n:\circ_n(b_n)\} \cap icols/b_{grp}}(q)} \quad (25)$$

$$\frac{icols \neq \varnothing \qquad \{a_1,\dots,a_n\} \setminus icols \neq \varnothing \qquad \{c_1,\dots,c_m\} = icols}{\boxed{a_1 \ldots a_n} \rightarrow \boxed{c_1 \ldots c_m}} \quad (26)$$

$$\frac{icols = \varnothing \qquad n > 1}{\boxed{a_1 \ldots a_n} \rightarrow \boxed{a_1}} \quad (27) \qquad \frac{a \notin icols \qquad \circledast \in \{ⓡ, \vec{\#}, \vec{\curlyvee}, \curlyvee\}}{\circledast_{a:\dots}(q) \rightarrow q} \quad (28)$$

Rewrite 18 prunes projection lists. Note how the right-hand side chooses a compact notation ($\cap$)—as mentioned in the discussion of Rewrite 9 on page 53—to express an intersection of the output column names that ignores the input column names during the matching.

Rewrite 19 replaces a Cartesian product by its first child $q_1$, if the second child $q_2$ neither changes the cardinality of the result, nor adds required columns to it. An *analogous rewrite* that swaps the roles of $q_1$ and $q_2$ exists for Rewrite 19 (and many others). Here, we omit these straightforward variants as they provide no additional insight.[5]

Rewrite 20 trades an equi-join for its child $q_2$, if no column of $q_2$ is required in the downstream plan and, for each row in $q_1$, there is exactly one match in $q_2$.

[5]In Pathfinder, both the rewrite and its dual variant are implemented.

The check on *icols* ensures the former condition, while the interaction between keys and domains, as discussed on page 58, guarantees the latter.

Based on functional dependencies, Rewrite 21 prunes unreferenced columns that functionally depend on a required column. Similarly, Rewrite 22 makes use of the constant property. Rewrites 23 to 27 all prune unreferenced columns.

Rewrite 28 removes operators, if their result column is not referenced in the downstream plan. The rewrite plays an important role with respect to the minimization of order constraints. Whenever order is not required, for example, because of the subsequent application of aggregates, XPath location steps, or XQuery's `fn:unordered` function, the position information of the subplan is ignored, which leads to the removal of numbering operators that implement order maintenance [61].

**Performing projection pushdown.** Figure 4.1(a) shows the query plan of Query $Q_3$ annotated with the required column property.[6] At various places (annotations ①–④) operators introduce unreferenced columns.

Rewrite 28 removes the position maintenance at ① and ②, Rewrite 18 removes the unreferenced pos column at ③, and Rewrite 28 again prunes the last superfluous operator at ④. Figure 4.1(b) shows the result of the rewrite process. Because of the first round of rewrites, the *cols* information at annotation ② changed. Subsequently, Rewrite 4, which prunes the projection operator above ②, is triggered.

After the simplification, two adjacent projection operators still exist at annotation ①. While they may be merged by a more advanced variant of Rewrite 5, the common subplan elimination phase described in the next section removes most projections as a side effect of detecting common subplans.

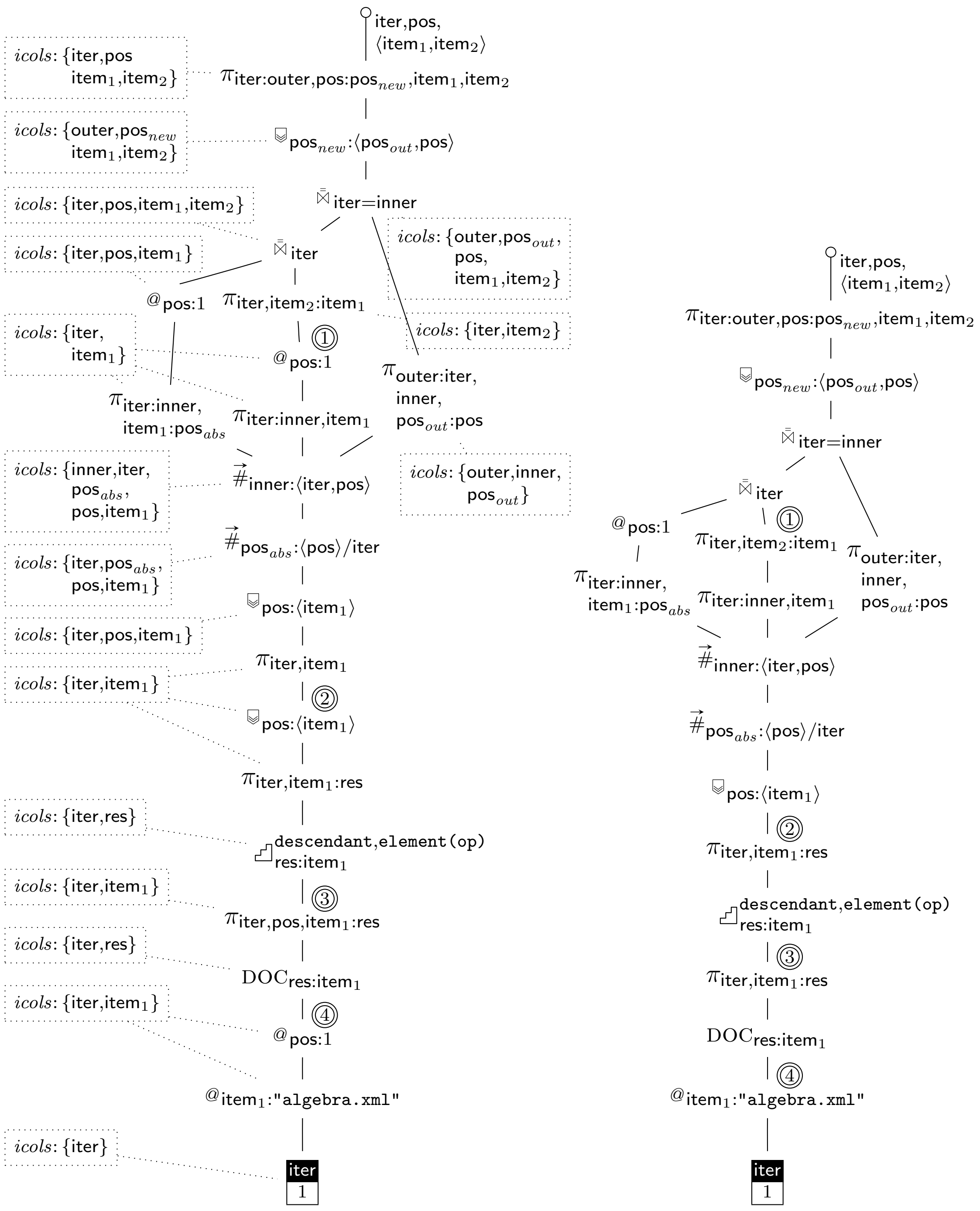
### 4.2.5 Common Subplan Elimination

Although the loop lifted compilation shares plan fragments whenever possible, the query text often contains redundant information. In the XQuery benchmark XMark, for example, most queries perform a subset of their path steps multiple times. Because of its materialization strategy, MonetDB/XQuery can exploit additional sharing—most probably, this applies to other database systems as well.

Here, we build on the ideas of global common subexpression elimination in compiler construction [30] and provide a global simplification—*common subplan elimination*—that is able to detect identical subplans. The common subplan elimination replaces all duplicate subplans by a reference to the initial matching subplan root—that is, a DAG edge is introduced.

Figure 4.2 and Table 4.12 describe a common subplan elimination algorithm `cse`. The algorithm ignores column names and thus—as a side effect—provides new

[6]Every operator has its own *icols* property information attached. We, however, summarized adjacent equivalent properties to increase the readability.



**(a)** Query plan annotated with *icols* property. **(b)** Simplified query plan.

Figure 4.1: The effect of *icols*-based projection pushdown: The query plan encoding Query $Q_3$ with *icols* property annotations in (a) is rewritten into the query plan in (b) by applying Rewrite 28 at places ①, ②, ④ and Rewrite 18 at ③.

```
cse($\circledast$)
    (op-list, op, proj) = traverse($\perp$, $\circledast$)
    return op                                        $\langle$extract the new $\varphi$ operator$\rangle$

traverse(op-list, $\circledast$)                               $\langle$traverse the subplan of $\circledast$$\rangle$
    if $\circledast$ has two children $q_1$ and $q_2$
        (op-list', $op_1$, $proj_1$) = traverse(op-list, $q_1$)
        (op-list'', $op_2$, $proj_2$) = traverse(op-list', $q_2$)
    else if $\circledast$ has one child q
        (op-list'', op, proj) = traverse(op-list, q)
    else
        op-list'' = op-list
    ($\circledast$, $\circledast_{cse}$, $\pi_{cse}$) $\in$ Table 4.12                   $\langle$look up $\circledast_{cse}$ and $\pi_{cse}$$\rangle$
    return ref(op-list'', $\circledast_{cse}$, $\pi_{cse}$)   $\langle$collect the correct reference of $\circledast_{cse}$$\rangle$

ref(op-list, $\circledast$, proj)                $\langle$traverse op-list and find a match for $\circledast$$\rangle$
    if op-list is $\perp$                                            $\langle$no matches found$\rangle$
        for each $*_i$ in $\circledast$           $\langle$assign new column names to $\circledast$ and proj$\rangle$
            generate new column name $c_i$
            replace $*_i$ in $\circledast$ by $c_i$
            replace $*_i$ in proj by $c_i$
        return (($\circledast$, $\perp$), $\circledast$, proj)                $\langle$use $\circledast$ as new representative$\rangle$
    else
        (op, op-list') = op-list                                        $\langle$split up op-list$\rangle$
        if op matches $\circledast$
            for each $*_i$ in $\circledast$                                       $\langle$adjust proj$\rangle$
                find corresponding column $c_i$ in op
                replace $*_i$ in proj by $c_i$
            return (op-list, op, proj)            $\langle$apply cse – reuse operator op$\rangle$
        else                      $\langle$no match – traverse the remainder of the list$\rangle$
            (op-list'', $\circledast$', proj') = ref(op-list', $\circledast$, proj)
            return ((op, op-list''), $\circledast$', proj')
```

Figure 4.2: Common subplan elimination algorithm cse.

| $\circledast$ | $\circledast_{cse}$ | $\pi_{cse}$ |
|---|---|---|
| $\pi_{a_1:b_1,\ldots,a_n:b_n}$ | $\mathsf{op}$ | $[\![\{a_1\!:\!b_1,\ldots,a_n\!:\!b_n\}]\!]_{\mathsf{proj}}$ |
| $\sigma_b$ | $\sigma_{[\![b]\!]_{\mathsf{proj}}}(\mathsf{op})$ | $\mathsf{proj}$ |
| $\times$ | $\mathsf{op}_1 \times \mathsf{op}_2$ | $\mathsf{proj}_1 \cup \mathsf{proj}_2$ |
| $\bar{\bowtie}_{b_1=b_2}$ | $\mathsf{op}_1 \; \bar{\bowtie}_{[\![b_1]\!]_{\mathsf{proj}_1}=[\![b_2]\!]_{\mathsf{proj}_2}} \; \mathsf{op}_2$ | $\mathsf{proj}_1 \cup \mathsf{proj}_2$ |
| $\bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}$ | $\mathsf{op}_1 \bowtie_{[\![b_{1\cdot 1}]\!]_{\mathsf{proj}_1}\theta_1[\![b_{2\cdot 1}]\!]_{\mathsf{proj}_2}\wedge\cdots\wedge [\![b_{1\cdot n}]\!]_{\mathsf{proj}_1}\theta_n[\![b_{2\cdot n}]\!]_{\mathsf{proj}_2}} \mathsf{op}_2$ | $\mathsf{proj}_1 \cup \mathsf{proj}_2$ |
| $\delta$ | $\delta\,(\pi_{[\![cols]\!]_{\mathsf{proj}}}(\mathsf{op}))$ | $\mathsf{proj}$ |
| $\uplus_{b_1,\ldots,b_n}$ | $(\pi_{[\![*_1:b_1,\ldots,*_n:b_n]\!]_{\mathsf{proj}_1}}(\mathsf{op}_1)) \uplus (\pi_{[\![*_1:b_1,\ldots,*_n:b_n]\!]_{\mathsf{proj}_2}}(\mathsf{op}_2))$ | $\{b_1\!:\!*_1,\ldots,b_n\!:\!*_n\}$ |
| $\setminus_{b_1,\ldots,b_n}$ | $(\pi_{[\![*_1:b_1,\ldots,*_n:b_n]\!]_{\mathsf{proj}_1}}(\mathsf{op}_1)) \setminus (\pi_{[\![*_1:b_1,\ldots,*_n:b_n]\!]_{\mathsf{proj}_2}}(\mathsf{op}_2))$ | $\{b_1\!:\!*_1,\ldots,b_n\!:\!*_n\}$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}$ | $\mathrm{GRP}_{[\![*_1:\circ_1(b_1),\ldots,*_n:\circ_n(b_n)/b_{grp}]\!]_{\mathsf{proj}}}(\mathsf{op})$ | $\{a_1\!:\!*_1,\ldots,a_n\!:\!*_n\}$ |
| $\begin{array}{\|c\|}\hline a_1 \ldots a_n \\ \hline \varnothing \\ \hline\end{array}$ | $\begin{array}{\|c\|}\hline *_1 \ldots *_n \\ \hline \varnothing \\ \hline\end{array}$ | $\{a_1\!:\!*_1,\ldots,a_n\!:\!*_n\}$ |
| $\begin{array}{\|c\|c\|c\|}\hline a_1 & \ldots & a_n \\ \hline & & \\ \hline\end{array}$ | $\begin{array}{\|c\|c\|c\|}\hline *_1 & \ldots & *_n \\ \hline & & \\ \hline\end{array}$ | $\{a_1\!:\!*_1,\ldots,a_n\!:\!*_n\}$ |
| $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|l\|}{name} \\ \hline a_1 & \ldots & a_n \\ \hline\end{array}$ | $\begin{array}{\|c\|c\|c\|}\hline \multicolumn{3}{\|l\|}{name} \\ \hline *_1 & \ldots & *_n \\ \hline\end{array}$ | $\{a_1\!:\!*_1,\ldots,a_n\!:\!*_n\}$ |
| $ⓡ_{a:\langle b_1,\ldots,b_n\rangle}$ | $ⓡ_{[\![*_1:\langle b_1,\ldots,b_n\rangle]\!]_{\mathsf{proj}}}(\mathsf{op})$ | $\{a\!:\!*_1\} \cup \mathsf{proj}$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(\mathsf{op})$ | $\vec{\#}_{[\![*_1:\langle b_1,\ldots,b_n\rangle/b_{grp}]\!]_{\mathsf{proj}}}(\mathsf{op})$ | $\{a\!:\!*_1\} \cup \mathsf{proj}$ |
| $\vec{\varrho}_{a:\langle b_1,\ldots,b_n\rangle}$ | $\vec{\varrho}_{[\![*_1:\langle b_1,\ldots,b_n\rangle]\!]_{\mathsf{proj}}}(\mathsf{op})$ | $\{a\!:\!*_1\} \cup \mathsf{proj}$ |
| $\varrho_{a:\langle b_1,\ldots,b_n\rangle}$ | $\varrho_{[\![*_1:\langle b_1,\ldots,b_n\rangle]\!]_{\mathsf{proj}}}(\mathsf{op})$ | $\{a\!:\!*_1\} \cup \mathsf{proj}$ |
| $⌟^{\alpha,v}_{a:\langle b\rangle}$ | $⌟^{\alpha,v}_{[\![*_1:\langle b\rangle]\!]_{\mathsf{proj}}}(\mathsf{op})$ | $\{a\!:\!*_1\} \cup \mathsf{proj}$ |
| $♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}\,(q)$ | $♀_{[\![b_1,b_2,\langle b_3,\ldots,b_n\rangle]\!]_{\mathsf{proj}}}(\mathsf{op})$ | — |

Table 4.12: Look-up table used by function `traverse` in Figure 4.2 that maps the input operators $\circledast$ to new operators $\circledast_{cse}$ and their corresponding projection lists $\pi_{cse}$. $\mathsf{op}$ and $\mathsf{proj}$ in the above table refer to the local variables in function `traverse`. $\circledast_{cse}$ and $\pi_{cse}$ introduce wildcard column names $*_1,\ldots,*_n$ and apply $[\![args]\!]_{proj}$ to rename the arguments $args$ based on the projection list $proj$.

column names and removes most projection operators. Algorithm `cse` traverses the given algebra plan (function `traverse`). For every operator, `traverse` returns a subplan without duplicate operators (op) and a projection list mapping the newly assigned column names to the original ones (proj).

The duplicate subplan removal is performed by function `ref`. The function consumes a duplicate-free list of operators (op-list), an operator blueprint ($\circledast_{cse}$), and its corresponding projection list ($\pi_{cse}$). The input operator and the projection list are generated by a lookup in Table 4.12. For each original operator $\circledast$, an equivalent representation that consumes the (now) distinct children operators is build.

To correctly cope with column names, Table 4.12 introduces the notion of $[\![args]\!]_{proj}$ that renames the *input* columns in *args* based on the projection list *proj*. The renaming of the following projection list $[\![\{\mathsf{a}:\mathsf{b},\mathsf{c}\}]\!]_{\{\mathsf{a}:\mathsf{item}_1,\mathsf{b}:\mathsf{item}_2,\mathsf{c}:\mathsf{item}_3\}}$, for example, matches columns b and c, thus leading to $\{\mathsf{a}:\mathsf{item}_2,\mathsf{c}:\mathsf{item}_3\}$.

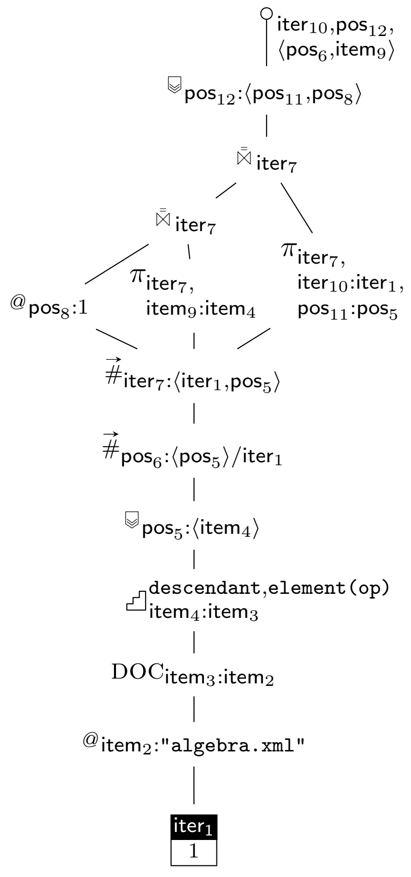


Figure 4.3: Query $Q_3$ after common subplan elimination. (Input to function `cse` was the query plan in Figure 4.1(b).)

To express the independence of column names, Table 4.12 furthermore introduces wildcard column names $*_1, \ldots, *_n$ for new *output* column names. The wildcard column names $*_i$ in $\circledast_{cse}$ and $\pi_{cse}$ are linked in terms of their index $i$. Because of the wildcard column names, the resulting operators $\circledast_{cse}$ are merely blueprints that ease the matching process of function `ref` in Figure 4.2.

Function `ref` recursively traverses the distinct operator list op-list (Lines 29–30) and generates a new distinct operator from $\circledast$, if no matching operator was found (Lines 16–20). Line 23 of Figure 4.2 checks, if an already existing operator op has the same children and characteristics as the blueprint operator $\circledast$. In this comparison the wildcard column names $*_i$ are ignored. In case of a successful match, the wildcard names in the projection list proj are replaced by the corresponding column names (Lines 24–26) and a reference to operator op is returned.

Figure 4.3 shows the result of applying common subplan elimination to the query plan of Figure 4.1(b). The only visible effects are the removal of four projection operators and the renaming of the column names. Naming conflicts at the equi-join operators led to the introduction of two additional projections, which provide new column names.

The algorithm in Figure 4.2 and Table 4.12

regards the input plan as a tree. Treating the input query plan as a DAG significantly reduces the optimization time of the common subplan elimination, yet increases the complexity of the algorithm. The algorithm furthermore ignores name collisions for the binary operators $\times$, $\bar{\bar{\bowtie}}$, and $\bowtie$. Whenever both children operators refer to the same column names a renaming projection has to be introduced.

The current algorithm can detect structurally equivalent plans only. Two adjacent operators with identical characteristics, for example, performing the same comparison, are not combined, however. Although such equivalences are detected by most SQL database back-ends, MonetDB/XQuery will evaluate the redundant operations. For operators $\textcircled{r}$, $\vec{\#}$, $\vec{\Cup}$, and $\Cup$ an extension of algorithm `cse` might take their subtrees into account: A modified algorithm can search for operators with identical characteristics up to the point where the semantics of the operators changes—$\vec{\#}$ and $\vec{\Cup}$ for example, require the input cardinality to stay unchanged.

## 4.2.6 Related Work

The simplification of loop lifted algebra plans discussed so far, are linked to various well-known properties and optimization techniques. We integrate many ideas on logical optimizations of Jarke and Koch's survey on query optimization in database systems [72, Section 3 Query Transformation].

To exemplify, in [72, Table 3], Jarke and Koch sketch methods that simplify *empty relations* at runtime. With the empty-input property *empty* and Rewrites 1 to 3, we are able to achieve a similar effect for all statically empty relations at compile time.

Based on transitivity laws *constant propagation* is able to detect and remove constant predicates [72]. Constant predicates are used to reduce the number of lines in a tableau [7]—this reduces the number of join predicates, effectively. The constant property *const* provides a similar way of constant propagation and Rewrites 6 to 8 remove the constant predicates.

The query rewrite facility in Starburst [99] provides several rules to get rid of duplicate elimination operations. [99, Rule 2. Distinct Pullup] describes an incarnation of Rewrite 11 taking a *one-tuple-condition* or a *quantifier-nodup-condition* into account. The former condition corresponds to the single-row property *card1*, while the latter is a variant of the *key* property. Here, the cardinality information of the single-row property extends the key information and thus Rewrite 11 needs to consider the keys only.

The domain property *dom* provides a data flow analysis similar to the active domains in [79]. Rewrite 13, which is based on domain inclusion, could be seen as an instance of an *unsatisfiable predicate* discussed in [72].

The functional dependency propagation for most of our algebra operators is equivalently described in [34]. The operators subsumed by $\textcircled{r}$, for example, correspond to the *extension* operator. [113] extends the property inference of [34] and provides key and functional dependency property explanations that directly match parts of our key and functional dependency properties [113, Section 5.2.1]. The functional dependency inference of [113] is furthermore extended by [125], whose additional observations for equi-joins (*Identity 3.1* and *Identity 3.2*) are depicted by the separate treatment of $\overset{=}{\bowtie}$ and $\bowtie$ in Table 4.8.

Sections 4.2.3 and 4.2.4 already gave an account of how our rewrites relate to projection and selection pushdown as described in [72]. The *idempotency rules* in [72, Rules M4 a)–j), Table2] are supposed to eliminate *common subexpressions*. Looking at our set of simplifying rewrites, these rules—especially Rules M4 d), g), and h)—relate to the combination of the Boolean required value property *req* and Rewrites 15 and 16.

With respect to MonetDB/XQuery, the required column dependency analysis bears some close resemblance with the dead code (or dead variable) elimination found in programming language compilers [35]: For every column, the code generator for MonetDB/XQuery generates procedural-style assignments that get evaluated regardless of their later usage. Unreferenced code fragments thus correspond to dead program code.

The common subplan elimination is a more explicit variant of *share equivalence* [92]. Both, share equivalent plan construction and the common subplan elimination synthesize the plans bottom up and ignore column renaming. Thanks to additional removal of projections, our common subplan elimination is less restrictive. Two projections sharing the same subplan are always share equivalent regardless of the number of projected columns.

## 4.3 Order Minimization

The second heuristic provides an unusual view on order optimization. In contrast to all other approaches we are aware of, we optimize ordering constraints on an unordered logical algebra. Our means to express order are the numbering operators $\vec{\#}$, $\vec{⛉}$, and ⛉, which we treat as first class algebra citizens.

The loop lifted compilation strategy ensures the correct order representation at any point in the query plan. In relational back-ends, however, these numbering operators enforce a physical order and thus restrict the back-ends in their freedom to optimize the execution plans. In what follows, we reduce and simplify the order constraints as much as possible. Any physical order optimization is orthogonal and may be performed by the back-end in addition.

### 4.3.1 Order Simplification

In Section 4.2.4 we introduced Rewrite 28, which removes any unreferenced numbering operator. This rule is especially effective in queries that feature grouping, aggregates, or path steps, as these language concepts ignore the sequence order of their inputs [61].

Rewrite 29 provides an equally important rule. The rewrite replaces a rank operator $\Cup$ by a renaming projection whenever a single column $b$ describes the ordering.[7]

$$\frac{}{\Cup_{a:\langle b\rangle}(q) \rightarrow \pi_{a:b,q.cols}(q)}\,(29)$$

The semantics of $\Cup$—column $a$ may be populated with values of any domain as long as they reflect the ranking criteria—ensures the correctness of this rewrite. A similar rewrite, however, is incorrect for the numbering operators $\vec{\#}$ and $\vec{\Cup}$.

In addition to the previous two house cleaning rewrites for order constraints, Rewrites 30 to 36 provide further simplifications for the numbering operators that take various properties into account.

$$\frac{\mathit{card1} \qquad \circledast \in \{\vec{\#}, \vec{\Cup}, \Cup\}}{\circledast_{a:\,\ldots}(q) \rightarrow @_{a:1}(q)}\,(30) \qquad \frac{\circledast \in \{\vec{\#}, \vec{\Cup}, \Cup\} \qquad \mathit{const} \cap \{b_1,\ldots,b_n\} \neq \varnothing}{\circledast_{a:\langle b_1,\ldots,b_n\rangle\,\ldots}(q) \rightarrow \circledast_{a:\langle b_1,\ldots,b_n\rangle \setminus \mathit{const}\,\ldots}(q)}\,(31)$$

$$\frac{\circledast \in \{\vec{\#}, \vec{\Cup}, \Cup\} \qquad \{b_i\} \in \mathit{key} \qquad i < n}{\circledast_{a:\langle b_1,\ldots,b_n\rangle\,\ldots}(q) \rightarrow \circledast_{a:\langle b_1,\ldots,b_i\rangle\,\ldots}(q)}\,(32)$$

$$\frac{\circledast \in \{\vec{\#}, \vec{\Cup}, \Cup\} \qquad (b_i \rightarrow b_j) \in \mathit{fd} \qquad i < j}{\circledast_{a:\langle b_1,\ldots,b_n\rangle\,\ldots}(q) \rightarrow \circledast_{a:\langle b_1,\ldots,b_{j-1},b_{j+1},\ldots,b_n\rangle\,\ldots}(q)}\,(33) \qquad \frac{\circledast \in \{\vec{\Cup}, \Cup\}}{\circledast_{a:\langle\rangle}(q) \rightarrow @_{a:1}(q)}\,(34)$$

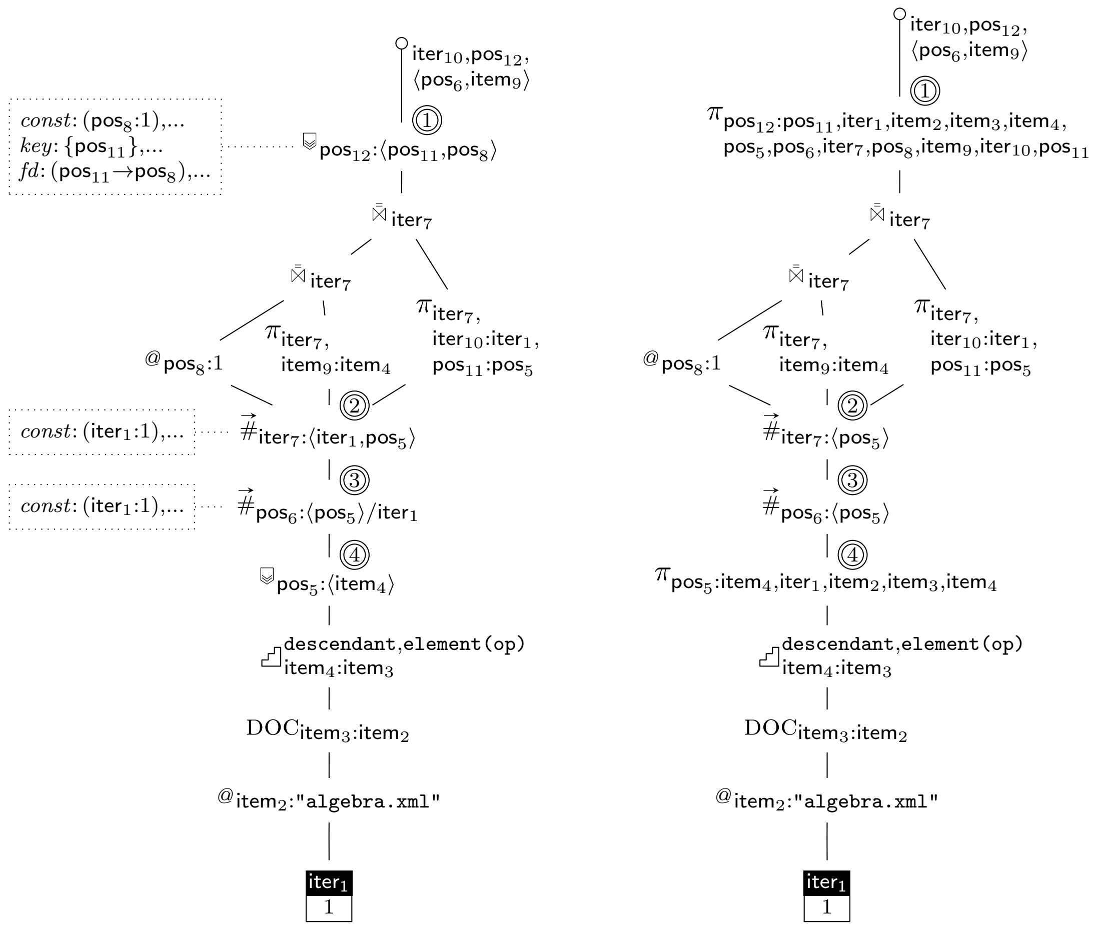
$$\frac{\{b_{grp}\} \in \mathit{key}}{\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle / b_{grp}}(q) \rightarrow @_{a:1}(q)}\,(35) \qquad \frac{(b_{grp}, v) \in \mathit{const}}{\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle / b_{grp}}(q) \rightarrow \vec{\#}_{a:\langle b_1,\ldots,b_n\rangle}(q)}\,(36)$$

Rewrites 31 to 33 consider constant and key columns as well as functional dependencies, to prune the list of ordering criteria. Rewrites 35 and 36 analyze the grouping column $b_{grp}$ of a $\vec{\#}$ operator and eliminate the operator or the grouping column.

Figure 4.4(a) shows an annotated variant of the query plan of Query $Q_3$ after common subplan elimination (Figure 4.3). The annotations list property entries that might be used during the order simplifications.[8] At annotation ④, Rewrite 29 trades the rank operator for a projection (Figure 4.4(b)). Rewrites 31 and 36 use the constant information to prune arguments of the row numbering operators at ② and ③. The argument of the rank operator at annotation ① can be simplified by either Rewrite 31, Rewrite 32, or Rewrite 33. The inferred properties enable

[7]For ease of presentation we omitted the order direction for the sort criteria. (Most rewrites retain the order directions. Exceptions are Rewrite 29, which requires an ascending order direction to perform the rewrite, Rewrite 38, which introduces ascending directions, and Rewrite 40, which needs to reverse the order of criteria $c_1, \ldots, c_m$ in case the order direction of $b_i$ is descending.)

[8]Annotating the properties of all operators is impractical—the $\vec{\#}$ operator at ② in Figure 4.4(a), for example, carries the information about 2 constant columns, 7 keys, and 10 functional dependencies.



**(a)** Cleaned up query plan of Query $Q_3$ (Figure 4.3) with property annotations.

**(b)** Query plan of Query $Q_3$ after order simplifications (at ①–④).

Figure 4.4: The effect of order simplification.

any of the three rewrites to prune column $\mathsf{pos}_8$. The resulting rank operator with a single ordering criterion is again subject to Rewrite 29.

## 4.3.2 Numbering Operator Conversion

The different ordering operators relate to each other. A row numbering operator $\vec{\#}$ without grouping criterion can be safely replaced by a row ranking operator $\vec{\varrho}$, if the sorting criteria either form a key or can be extended to form a key. Row ranking operators $\vec{\varrho}$ furthermore can be transformed into ranking operators $\varrho$, if their result is used only as order, grouping, or distinct criterion—that is, if the generated values are immaterial.

Converting the ordering operators into less restrictive variants allows the peephole-style optimization to better exploit the operator characteristics. In addition to Rewrite 29, for example, Section 4.3.3 introduces more rewrites that can benefit from such a conversion.

Rewrites 37 to 39 exploit the relationship between the different ordering operators. The usage property *use* (Table 4.13), which forms the pre-condition in Rewrite 39, is inferred top-down and records if a column's actual values are required to provide the correct result.

$$\frac{k \in \mathit{key} \qquad k \cap \{b_1, \ldots, b_n\} = k}{\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle}(q) \rightarrow \vec{\curlyveedownarrow}_{a:\langle b_1,\ldots,b_n\rangle}(q)}(37) \qquad \frac{k \in \mathit{key}}{\vec{\#}_{a:\langle\rangle}(q) \rightarrow \vec{\curlyveedownarrow}_{a:\langle k\rangle}(q)}(38)$$

$$\frac{a \notin \mathit{use}}{\vec{\curlyveedownarrow}_{a:\langle b_1,\ldots,b_n\rangle}(q) \rightarrow \curlyveedownarrow_{a:\langle b_1,\ldots,b_n\rangle}(q)}(39)$$

Rewrite 37, for example, transforms the $\vec{\#}$ operators at annotations ② and ③ in Figure 4.4(b). The house cleaning rewrites described in Section 4.2 might further simplify the plan in Figure 4.4(b) by removing the unreferenced $@_{\mathsf{pos}_8:1}$ operator and the projections at ① and ④.[9]

### 4.3.3 Order Pushup

The compilation scheme described in Chapter 2 uses rank operators $\curlyveedownarrow$ to correctly maintain the sequence order of the input query. The absolute sequence positions, however, can be observed only by accessing the designated sequence position values (Rule OrderedFor in Section 2.4). These absolute position values are provided by a row numbering operator $\vec{\#}$. Similarly, the generation of surrogate values is achieved by $\vec{\#}$ and $\vec{\curlyveedownarrow}$ operators. The sequence order maintained by $\curlyveedownarrow$ operators merely yields the correct ordering criteria for these numbering operators.

To minimize the order-maintaining rank operators, Rewrites 40 to 45 push up the $\curlyveedownarrow$ operators through almost *any* other operator. Rewrite 40 replaces a rank's result column by its ordering criteria whenever possible. The helper function *cols*($p$) used in Rewrites 42 and 43 determines the columns used in argument $p$.[10] The Rewrites 40 to 45 are applied until operator $\curlyveedownarrow$ is either not referenced anymore (and gets pruned by Rewrite 28), gets stuck underneath a $\dot{\cup}$ or GRP operator, or reaches the plan root ⚲. A rank operator may pass all filtering operators ($\sigma$, $\bar{\ltimes}$, $\ltimes$, and ⏍) as it may provide arbitrary rank values—regardless of domain or density. As the ordering criteria of a rank operator functionally depend on the result column, operator $\curlyveedownarrow$ furthermore does not affect the result of a distinct operator.

$$\frac{\circledast \in \{\vec{\#}, \vec{\curlyveedownarrow}, \curlyveedownarrow\}}{\circledast_{a:\langle\ldots,b_{i-1},b_i,b_{i+1},\ldots\rangle \ldots}\left(\curlyveedownarrow_{b_i:\langle c_1,\ldots,c_m\rangle}(q)\right) \rightarrow \circledast_{a:\langle\ldots,b_{i-1},c_1,\ldots,c_m,b_{i+1},\ldots\rangle \ldots}\left(\curlyveedownarrow_{b_i:\langle c_1,\ldots,c_m\rangle}(q)\right)}(40)$$

[9] Peek forward to Figure 4.5(a) (page 82) to observe these effects.

[10] The compilation rules in Chapter 2 ensure that condition $a \notin \mathit{cols}(p)$ always holds (if Rewrite 40 is performed before Rewrite 42).

| **Usage Property** *use* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}(q)$ | $q.use \leftarrow q.use \cup \{b_i \mid a_i \in use\}$ |
| $\sigma_b(q)$ | $q.use \leftarrow q.use \cup \{b\} \cup use$ |
| $\times(q_1,q_2)$ | $q_1.use \leftarrow q_1.use \cup \{c \mid c \in q_1.cols, c \in use\}$ |
| $\times(q_1,q_2)$ | $q_2.use \leftarrow q_2.use \cup \{c \mid c \in q_2.cols, c \in use\}$ |
| $\bar{\bowtie}_{b_1=b_2}(q_1,q_2)$ | $q_1.use \leftarrow q_1.use \cup \{b_1\} \cup \{c \mid c \in q_1.cols, c \in use\}$ |
| $\bar{\bowtie}_{b_1=b_2}(q_1,q_2)$ | $q_2.use \leftarrow q_2.use \cup \{b_2\} \cup \{c \mid c \in q_2.cols, c \in use\}$ |
| $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}(q_1,q_2)$ | $q_1.use \leftarrow q_1.use \cup \{b_{1\cdot i} \mid i \in \{1,\ldots,n\}\} \cup \{c \mid c \in q_1.cols, c \in use\}$ |
| $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\wedge\cdots\wedge b_{1\cdot n}\theta_n b_{2\cdot n}}(q_1,q_2)$ | $q_2.use \leftarrow q_2.use \cup \{b_{2\cdot i} \mid i \in \{1,\ldots,n\}\} \cup \{c \mid c \in q_2.cols, c \in use\}$ |
| $\delta(q)$ | $q.use \leftarrow q.use \cup use$ |
| $\dot{\cup}_{b_1,\ldots,b_n}(q_1,q_2)$, $\setminus_{b_1,\ldots,b_n}(q_1,q_2)$ | $q_1.use \leftarrow q_1.use \cup \{b_i \mid i \in \{1,\ldots,n\}\}$ |
| $\dot{\cup}_{b_1,\ldots,b_n}(q_1,q_2)$, $\setminus_{b_1,\ldots,b_n}(q_1,q_2)$ | $q_2.use \leftarrow q_2.use \cup \{b_i \mid i \in \{1,\ldots,n\}\}$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}(q)$ | $q.use \leftarrow q.use \cup \{b_i \mid i \in \{1,\ldots,n\}\} \cup \{b_{grp} \mid b_{grp} \in use\}$ |
| $\textcircled{r}_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle}(q)$ | $q.use \leftarrow q.use \cup \{b_i \mid i \in \{1,\ldots,n\}\} \cup \{c \mid c \in use, c \neq a\}$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(q)$, $\vec{\Downarrow}_{a:\langle b_1,\ldots,b_n\rangle}(q)$, $\Downarrow_{a:\langle b_1,\ldots,b_n\rangle}(q)$ | $q.use \leftarrow q.use \cup \{c \mid c \in use, c \neq a\}$ |
| $\llcorner^{\alpha,v}_{a:\langle b\rangle}(q)$ | $q.use \leftarrow q.use \cup \{b\} \cup \{c \mid c \in use, c \neq a\}$ |
| $♀_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}(q)$ | $q.use \leftarrow \{b_i \mid i \in \{1,3,\ldots,n\}\}$ |

Table 4.13: Top-down inference of the column usage property *use* that infers the set of columns whose values may not be modified to ensure the correct result. (*use* is seeded with $\emptyset$.)

$$\frac{}{\pi_{a,c_1,\dots,c_m}\left(\Cup_{a:\langle b_1,\dots,b_n\rangle}(q)\right) \rightarrow \Cup_{a:\langle b_1,\dots,b_n\rangle}\left(\pi_{b_1,\dots,b_n,c_1,\dots,c_m}(q)\right)} \quad (41)$$

$$\frac{\circledast \in \{\sigma_p, \delta, \textcircled{r}_p, \vec{\#}_p, \vec{\Cup}_p, \lrcorner_p^{\alpha,v}\} \qquad a \notin \mathit{cols}(p)}{\circledast(\Cup_{a:\langle b_1,\dots,b_n\rangle}(q)) \rightarrow \Cup_{a:\langle b_1,\dots,b_n\rangle}(\circledast(q))} \quad (42)$$

$$\frac{\circledast \in \{\times, \bar{\bar{\ltimes}}_p, \bowtie_p\} \qquad a \notin \mathit{cols}(p)}{\Cup_{a:\langle b_1,\dots,b_n\rangle}(q_1) \circledast q_2 \rightarrow \Cup_{a:\langle b_1,\dots,b_n\rangle}(q_1 \circledast q_2)} \quad (43)$$

$$\frac{}{\left(\Cup_{a:\langle b_1,\dots,b_n\rangle}(q_1)\right) \dot\cup_{a,c_1,\dots,c_m} \left(\Cup_{a:\langle b_1,\dots,b_n\rangle}(q_2)\right) \rightarrow \Cup_{a:\langle b_1,\dots,b_n\rangle}\left(q_1 \dot\cup_{b_1,\dots,b_n,c_1,\dots,c_m} q_2\right)} \quad (44)$$

$$\frac{(b_{grp} \rightarrow a) \in \Cup \qquad a \neq b_{grp}}{\textsc{grp}_{a:\textsc{the}(a),\dots/b_{grp}}\left(\Cup_{a:\langle b_1,\dots,b_n\rangle}(q)\right) \rightarrow \Cup_{a:\langle b_1,\dots,b_n\rangle}\left(\textsc{grp}_{b_1:\textsc{the}(b_1),\dots,b_n:\textsc{the}(b_n),\dots/b_{grp}}(q)\right)} \quad (45)$$

**Workhorse Queries.** Flat queries that do not read positional variables, never invoke aggregate functions, and do not construct sequences will feature no $\vec{\#}$, $\vec{\Cup}$, GRP, and $\dot\cup$ operators. An XQuery dialect with these characteristics is XQuery's data-bound "workhorse", a term we coined in [54]. In such a language fragment, the rewrites ensure the *complete* order push up where at most a single rank operator in the plan tail—right underneath the plan root $\varphi$—remains.

Instead of applying Rewrite 29 at ④ in Figure 4.4(a), the sequence of Rewrites 40, 42, 40, 42, 28, 28, 41, 43, 40, and 28 could have pushed the $\Cup$ operator up in the plan, leading to a similar effect: All references of its result columns are replaced (Rewrite 40) and the operator can be eliminated (Rewrite 28).

## 4.3.4 Related Work

Since the seminal work on *interesting orders* [111], various approaches concentrated on order optimization [91, 113, 114, 125]. Similar approaches, in the context of XQuery, further underline the need for order optimization [41, 97, 124].

Except for [41], all of these approaches optimize the order based on an *ordered* algebra. Our optimizations, however, are based on an unordered logical algebra. The rewrites are not designed to yield an optimal execution plan. Instead they heuristically try to minimize the number of ordering operators. *Any* order optimization on a physical algebra is orthogonal and may, for example, push down the remaining order constraints.[11]

Simmen, Shekita, and Malkemus describe an approach to reduce the order constraints of interesting orders [113, Section 4.1]. They base their decision on constant columns, keys, and functional dependencies. Their order reduction removes ordering criteria similar to Rewrites 31 to 33. The effect of Rewrites 32 and 33 is also described by *Identities 1* and *4* of [125, Figure 4]. *Identity 3*

[11]We observed DB2 pushing down the remaining order constraints [54].

describes the effect of algorithm `Reduce Order` in [113] and might be supported in our setup by a more advanced inference of functional dependencies.

Wang and Cherniack furthermore take groupings into account, which could be interesting in our simplifications as well [125]. Currently we express a grouping with the help of the ordered $\vec{⩔}$ operator to guarantee stable surrogate values, which allow to link separate queries that together encode a nested result. In case the surrogate values are only visible inside a single query, we could, however, exploit the grouping optimizations described in *Identities 2* and *5* by means of a new grouping operator that behaves like a ⩔ operator without the guarantee of an ascending output order.

The order rewrites in [114, Section 4.3] push down a sort operator through all operators of an order-preserving algebra. Putting the direction of the rewrites aside, this approach relates to the rank pushup in Section 4.3.3. *Rewrite rule S8* for example, constitutes a variant of expressing the functional dependency for grouped aggregates (Rewrite 45). Whereas the ⩔ operator enables a similar order maintenance as the sort operator in [114], our algebra retains more freedom as its operators are completely independent of the back-end implementation.

The authors of [41] remove order constraints based on the XQuery Core representation—an effect achieved by Rewrite 28. Whereas their rewrites suffice to remove order constraints in XPath queries, the principal data structure of XQuery—item sequences—prohibits merging of multiple orders as performed by Rewrite 40. Despite that, we might still benefit from the optimization in [41], as it takes more XPath step knowledge into account. Except for the property inference of keys and functional dependencies, we currently ignore the path step information and thus may end up ordering query results on more columns than strictly necessary.

An algebra over ordered tables is the subject of order optimization in [124]. An order context framework provides *minimal ordered semantics* by removing—much like Rewrite 28—superfluous *Sortby* operators. In addition, order is merged in join operators and pushed through the plan in an *Orderby Pullup* phase much like in Section 4.3.3. In the presence of *order-destroying* operators such as $\delta$, the technique of [124] however fails to propagate order information to the plan tail (compare with Rewrite 42).

The XML database system Timber [71] implements the TLC algebra, which manipulates sets of heterogeneous, ordered, label trees. Sequence order is supported by an extension of the tree algebra [97]. This algebra—coined TLC-C—consists of order-preserving logical operators and a sort operator. In comparison to the loop lifted compilation, order is added after the query translation, leading to plans whose order constraints are placed near the plan root—as in our rewritten query plans. In Timber, iteration order, sequence order, and document order is equivalent—note that this simplification of matters violates the XQuery semantics. Our compilation technique, on the other hand, can differentiate these different concepts. We can, for example, correctly treat XQuery's positional predicates, but can also exploit loosened order constraints, such as XQuery's ordering mode `unordered` [61].

## 4.4 Query Unnesting

The application of the house cleaning and order minimization heuristics already leads to a significant number of simplifications. The resulting query plans, however, still reflect the nested iteration semantics of the loop lifted compilation scheme. The numbering operators that encode the iteration values in the compilation rules' iter columns and the large number of aligning equi-joins along these iteration columns still enforce a particular evaluation order on the various back-ends.

In most cases, the compilation rules introduced the aligning equi-joins to combine columns of a table that were split earlier in the compilation process. This section's heuristic aims to get rid of the aligning equi-joins (Section 4.4.1) and subsequently unnest the nested query plans (Section 4.4.3). To support this heuristic, we furthermore relocate the duplicate elimination operators (Section 4.4.2). The $\delta$ operators are pushed up in the plan, leading to a more effective query unnesting and providing the back-ends with more freedom to perform join reordering [54].

### 4.4.1 Equi-Join Pushdown

The removal of aligning equi-join is performed by a sequence of peephole rewrites that push down an equi-join until it directly references the split point and thus can be eliminated. The rewrites replace one of the child operators $\circledast$ by its respective input and create a copy of $\circledast$ on top of the $\bar{\bowtie}$ operator. If the two input operators of an equi-join are identical, we may remove the $\bar{\bowtie}$ operator in many situations.

To avoid that an $\bar{\bowtie}$ is pushed to the bottom of a plan (e.g., first through all its left children and then through all its right children) before it references the same operator twice, we take the DAG structure of the query plan into account. We use $\Rrightarrow$ to denote the reachability relation of the DAG—♀ $\Rrightarrow \circledast$, for example, holds for all operators $\circledast$ in the query plan. Based on the reachability check $q_1 \not\Rrightarrow q_2$ we can prohibit that an equi-join $q_1 \bar{\bowtie} q_2$ is pushed through operator $q_2$ (and thus farther down than necessary). The relationship $q_1 \Rrightarrow q_2 \wedge q_2 \Rrightarrow q_1$ furthermore ensures the operator identity of $q_1$ and $q_2$.[12]

Rewrites 46 to 51 remove aligning equi-joins. These rewrites all reference the same node twice, as the reachability checks enforce that $q_1$ is identical to $q_2$. Rewrite 46 represents the simplest form of equi-join removal: an $\bar{\bowtie}$ operator on the key column $b$ is guaranteed to return the input unmodified and can be pruned. Rewrite 47 is a variation of Rewrite 46 with an additional renaming projection.

[12]For any operator $q_1 \neq q_2$, the relationship $q_1 \Rrightarrow q_2 \wedge q_2 \Rrightarrow q_1$ would incorrectly indicate a cycle in the DAG.

$$\frac{q_1 \Rrightarrow q_2 \qquad q_2 \Rrightarrow q_1 \qquad \{b\} \in \mathit{key}}{q_1 \mathbin{\bar{\bowtie}_b} q_2 \rightarrow q_1} \tag{46}$$

$$\frac{q_1 \Rrightarrow q_2 \qquad q_2 \Rrightarrow q_1 \qquad \{b_1\} \in q_2.\mathit{key}}{(\pi_{a_1:b_1,\ldots,a_n:b_n}(q_1)) \mathbin{\bar{\bowtie}_{a_1=b_1}} q_2 \rightarrow \pi_{\{a_1:b_1,\ldots,a_n:b_n\} \cup q_2.\mathit{cols}}(q_1)} \tag{47}$$

$$\frac{q_1 \Rrightarrow q_2 \qquad q_2 \Rrightarrow q_1 \qquad \mathit{set} \qquad \{(a_1 \rightarrow a_2), \ldots, (a_1 \rightarrow a_n)\} \subseteq \mathit{fd}}{(\pi_{a_1:b_1,\ldots,a_n:b_n}(q_1)) \mathbin{\bar{\bowtie}_{a_1=b_1}} q_2 \rightarrow \pi_{\{a_1:b_1,\ldots,a_n:b_n\} \cup q_2.\mathit{cols}}(q_1)} \tag{48}$$

$$\frac{q_1 \Rrightarrow q_2 \qquad q_2 \Rrightarrow q_1}{\left(\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(q_1)\right) \mathbin{\bar{\bowtie}_{b_{grp},a}} \left(\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(q_2)\right) \rightarrow \vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(q_1)} \tag{49}$$

$$\frac{q_1 \Rrightarrow q_2 \qquad q_2 \Rrightarrow q_1}{\left(\mathrm{GRP}_{p_1/b_{grp}}(q_1)\right) \mathbin{\bar{\bowtie}_{b_{grp}}} \left(\mathrm{GRP}_{p_2/b_{grp}}(q_2)\right) \rightarrow \mathrm{GRP}_{p_1,p_2/b_{grp}}(q_1)} \tag{50}$$

$$\frac{q_1 \Rrightarrow q_2 \qquad q_2 \Rrightarrow q_1 \qquad \mathit{set} \qquad \{(b_{grp} \rightarrow c_1), \ldots, (b_{grp} \rightarrow c_m)\} \subseteq q_2.\mathit{fd}}{\left(\mathrm{GRP}_{p/b_{grp}}(q_1)\right) \mathbin{\bar{\bowtie}_{b_{grp}}} \left(\pi_{b_{grp},c_1,\ldots,c_m}(q_2)\right) \rightarrow \mathrm{GRP}_{p,c_1:\mathrm{THE}(c_1),\ldots,c_m:\mathrm{THE}(c_m)/b_{grp}}(q_1)} \tag{51}$$

In case the query plans features an $\bar{\bowtie}$ operator joining along a non-key column, Rewrite 48 may remove the equi-join operator, if all columns functionally depend on the join column $a_1$ and the result is subject to duplicate elimination (*set*). The set property *set* (Table 4.14) is inferred in a top-down DAG traversal and marks all operators, whose result is affected by a $\delta$ operator in the downstream plan. The set property inference can be seen as a more modular representation of *Rule 3. Distinct Pushdown From/To* described in [99].

Rewrite 49 removes an equi-join that positionally aligns its inputs by means of a $\vec{\#}$ operator. Such an aligning $\bar{\bowtie}$ operator is, for example, introduced by Rule GROUP in Chapter 2. If the grouping columns and the ordering criteria of the numbering operators coincide, the join columns are aligned and form a key. Consequently Rewrite 49 prunes the equi-join.

The grouping operators in Rewrite 50 ensure the keyness of column $b_{grp}$. Rewrite 50 prunes the $\bar{\bowtie}$ operator and merges the arguments $p_1$ and $p_2$ of the GRP operators.[13] Rewrite 51 can be seen as the combination of Rewrite 50 and Rewrite 48 where the projection columns are transformed into THE aggregates.

Rewrites 52 to 61 push down an equi-join operator through its left input operator. We skipped the equivalent rewrites for the right-hand side input operators, as they add no new insight and may be easily constructed based on the existing rewrites. The combination of key and domain property checks in Rewrites 59 to 61 guarantee—as discussed on page 58—that the cardinality of the left input operator is not changed by the rewrites. The test on *#ref* in Rewrites 60 and 61 furthermore prohibits numbering operators to be split into two independent operators. Such a split leads to incomparable abstract active

[13]In this section we use $p$ to denote the arguments of an arbitrary operator.

| **Set Property** *set* | |
|---|---|
| $\pi_{a_1:b_1,..,a_n:b_n}(q),\ \sigma_b(q)$ | $q.set \ \leftarrow q.set \land set$ |
| $\times(q_1,q_2),\ \stackrel{=}{\bowtie}_{b_1=b_2}(q_1,q_2),$ $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\land\cdots\land b_{1\cdot n}\theta_n b_{2\cdot n}}(q_1,q_2)$ | $q_1.set \leftarrow q_1.set \land set$ |
| $\times(q_1,q_2),\ \stackrel{=}{\bowtie}_{b_1=b_2}(q_1,q_2),$ $\bowtie_{b_{1\cdot1}\theta_1 b_{2\cdot1}\land\cdots\land b_{1\cdot n}\theta_n b_{2\cdot n}}(q_1,q_2)$ | $q_2.set \leftarrow q_2.set \land set$ |
| $\delta(q)$ | $q.set \ \leftarrow q.set \land \text{‘}true$ |
| $\dot\cup_{b_1,\ldots,b_n}(q_1,q_2)$ | $q_1.set \leftarrow q_1.set \land set$ |
| $\dot\cup_{b_1,\ldots,b_n}(q_1,q_2)$ | $q_2.set \leftarrow q_2.set \land set$ |
| $\backslash_{b_1,\ldots,b_n}(q_1,q_2)$ | $q_1.set \leftarrow \text{‘}false$ |
| $\backslash_{b_1,\ldots,b_n}(q_1,q_2)$ | $q_2.set \leftarrow q_2.set \land \text{‘}true$ |
| $\mathrm{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}}(q)$ | $q.set \ \leftarrow \text{‘}false$ |
| $\textcircled{r}_{a:\langle b_1,\ldots,b_n\rangle}(q)$ | $q.set \ \leftarrow q.set \land set$ |
| $\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(q)$ | $q.set \ \leftarrow \text{‘}false$ |
| $\vec{\veebar}_{a:\langle b_1,\ldots,b_n\rangle}(q),\ \veebar_{a:\langle b_1,\ldots,b_n\rangle}(q),\ \sqcup^{\alpha,v}_{a:\langle b\rangle}(q)$ | $q.set \ \leftarrow q.set \land set$ |
| $\text{⚲}_{b_1,b_2,\langle b_3,\ldots,b_n\rangle}(q)$ | $q.set \ \leftarrow \text{‘}false$ |

Table 4.14: Top-down inference of the Boolean set property *set* that infers whether the output rows will undergo duplicate elimination in the upstream plan. (*set* is seeded with ‘*true*.)

domains for the two numbering operators and thus prohibits the application of Rewrites 59 to 61.

$$\frac{q_2 \nRightarrow \pi \qquad q_1.cols \cap q_2.cols = \varnothing}{(\pi_{a_1:b_1,p}(q_1)) \stackrel{=}{\bowtie}_{a_1=b_2} q_2 \rightarrow \pi_{\{a_1:b_1,p\}\cup q_2.cols}(q_1 \stackrel{=}{\bowtie}_{b_1=b_2} q_2)} \tag{52}$$

$$\frac{\circledast \in \{\sigma_p, \textcircled{r}_p, \veebar_p, \sqcup^{\alpha,v}_p\} \qquad q_2 \nRightarrow \circledast \qquad b_1 \in q_1.cols}{\circledast(q_1) \stackrel{=}{\bowtie}_{b_1=b_2} q_2 \rightarrow \circledast(q_1 \stackrel{=}{\bowtie}_{b_1=b_2} q_2)} \tag{53}$$

$$\frac{\circledast \in \{\times, \stackrel{=}{\bowtie}_p, \bowtie_p\} \qquad q_3 \nRightarrow \circledast \qquad b_2 \in q_2.cols}{(q_1 \circledast q_2) \stackrel{=}{\bowtie}_{b_2=b_3} q_3 \rightarrow q_1 \circledast (q_2 \stackrel{=}{\bowtie}_{b_2=b_3} q_3)} \tag{54}$$

$$\frac{q_2 \nRightarrow \delta \qquad k \in q_2.key}{\delta(q_1) \stackrel{=}{\bowtie}_{b_1=b_2} q_2 \rightarrow \delta(q_1 \stackrel{=}{\bowtie}_{b_1=b_2} q_2)} \tag{55}$$

$$\frac{q_3 \nRightarrow \dot\cup}{(q_1 \dot\cup_{b_1,p} q_2) \stackrel{=}{\bowtie}_{b_1=b_3} q_3 \rightarrow (q_1 \stackrel{=}{\bowtie}_{b_1=b_3} q_3) \dot\cup_{\{b_1,p\}\cup q_3.cols}(q_2 \stackrel{=}{\bowtie}_{b_1=b_3} q_3)} \tag{56}$$

$$\frac{q_3 \not\Rrightarrow \setminus \qquad \{b_3\} \in q_3.\mathit{key}}{(q_1 \setminus_{b_1,p} q_2) \overset{=}{\bowtie}_{b_1=b_3} q_3 \rightarrow (q_1 \overset{=}{\bowtie}_{b_1=b_3} q_3) \setminus_{\{b_1,p\}\cup q_3.\mathit{cols}} (q_2 \overset{=}{\bowtie}_{b_1=b_3} q_3)} \tag{57}$$

$$\frac{q_2 \not\Rrightarrow \text{GRP} \qquad \{b_2\} \in q_2.\mathit{key} \qquad \{c_1, \ldots, c_n\} = q_2.\mathit{cols}}{\left(\text{GRP}_{p/b_{grp}}\,(q_1)\right) \overset{=}{\bowtie}_{b_{grp}=b_2} q_2 \rightarrow \text{GRP}_{p,c_1:\text{THE}(c_1),\ldots,c_n:\text{THE}(c_2)/b_{grp}}\,(q_1 \overset{=}{\bowtie}_{b_{grp}=b_2} q_2)} \tag{58}$$

$$\frac{\begin{array}{c} q_2 \not\Rrightarrow \text{GRP} \qquad \{b_2\} \in q_2.\mathit{key} \qquad \{c_1, \ldots, c_n\} = q_2.\mathit{cols} \\ (b_1, d_1) \in q_1.\mathit{dom} \qquad (b_2, d_2) \in q_2.\mathit{dom} \qquad \mathit{subDom}\,(d_1, d_2) \end{array}}{\begin{array}{c} \left(\text{GRP}_{a:\text{THE}(b_1),p/b_{grp}}\,(q_1)\right) \overset{=}{\bowtie}_{a=b_2} q_2 \rightarrow \\ \text{GRP}_{a:\text{THE}(b_1),p,c_1:\text{THE}(c_1),\ldots,c_n:\text{THE}(c_2)/b_{grp}}\,(q_1 \overset{=}{\bowtie}_{b_1=b_2} q_2) \end{array}} \tag{59}$$

$$\frac{\begin{array}{c} \vec{\#}_p(q_1).\#\mathit{ref} = 1 \qquad \{b_2\} \in q_2.\mathit{key} \qquad b_1 \in q_1.\mathit{cols} \\ (b_1, d_1) \in q_1.\mathit{dom} \qquad (b_2, d_2) \in q_2.\mathit{dom} \qquad \mathit{subDom}\,(d_1, d_2) \end{array}}{\vec{\#}_p(q_1) \overset{=}{\bowtie}_{b_1=b_2} q_2 \rightarrow \vec{\#}_p(q_1 \overset{=}{\bowtie}_{b_1=b_2} q_2)} \tag{60}$$

$$\frac{\begin{array}{c} \vec{\Cup}_p(q_1).\#\mathit{ref} = 1 \qquad b_1 \in q_1.\mathit{cols} \\ (b_1, d_1) \in q_1.\mathit{dom} \qquad (b_2, d_2) \in q_2.\mathit{dom} \qquad \mathit{subDom}\,(d_1, d_2) \end{array}}{\vec{\Cup}_p(q_1) \overset{=}{\bowtie}_{b_1=b_2} q_2 \rightarrow \vec{\Cup}_p(q_1 \overset{=}{\bowtie}_{b_1=b_2} q_2)} \tag{61}$$

Rewrite 54 pushes an equi-join operator through another equi-join. Although this rewrite is useful in some situations, it may lead to an infinite loop where two adjacent $\overset{=}{\bowtie}$ operators are rewritten alternately. One possibility to resolve the infinite recursion is to split the rewrites into rounds. The $\overset{=}{\bowtie}$ operators are scheduled for rewrites based on their maximum distance from a leaf—that is, the bottom-most equi-join is pushed down first. If an $\overset{=}{\bowtie}$ operator cannot be pushed further down, it is suspended til the end of the round.

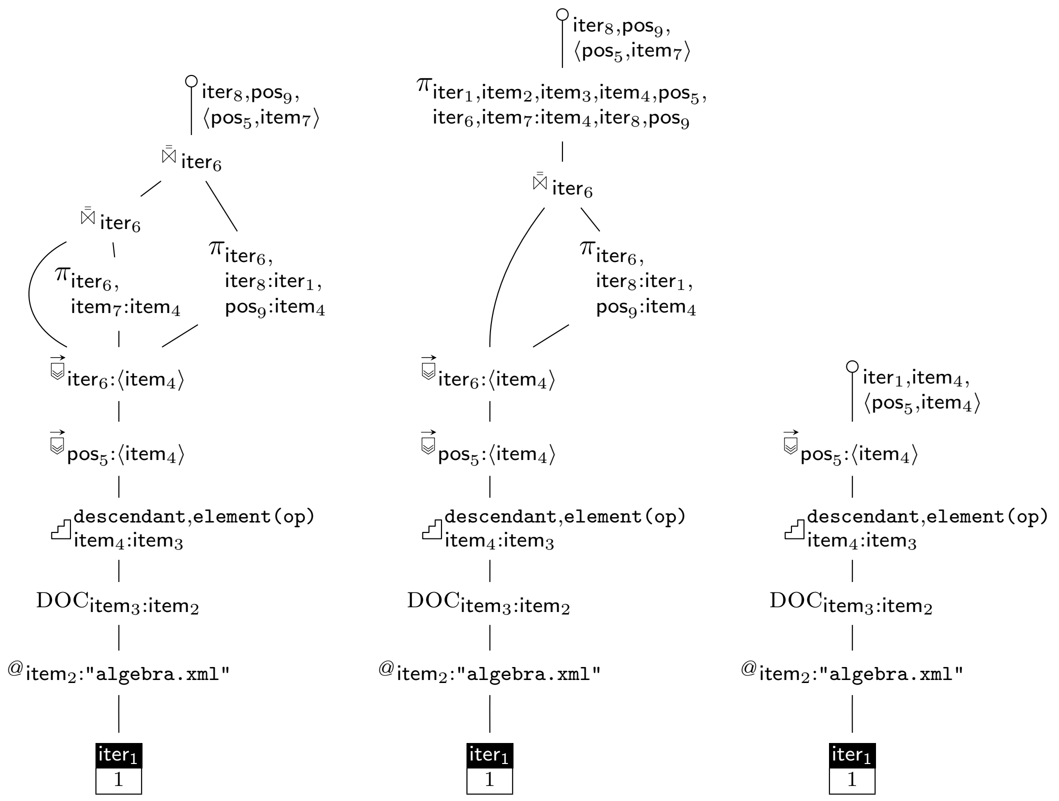
Starting from the query plan in Figure 4.5(a), Rewrite 47 removes the lower equi-join operator. Subsequently, Rewrite 52 pushes the upper equi-join through the projection resulting from the previous rewrite, before Rewrite 47 prunes the $\overset{=}{\bowtie}$ operator (Figure 4.5(b)). Algorithm `cse` and Rewrite 28 clean up the remaining projection operators and the now unreferenced numbering operator $\vec{\#}_{\mathsf{iter}_6:\langle\mathsf{item}_4\rangle}$ (Figure 4.5(c)). All remaining operators are necessary to correctly implement the query. In comparison to the initial query plan (Figure 2.9; page 23), more than 75 percent of the operators were pruned by the rewrites.

## 4.4.2 Distinct Operator Relocation

Excessive duplicate elimination slows down most database back-ends. With the *set* property inference (Table 4.14) we introduced a means to propagate the duplicate elimination information into an operator's complete subplan. Because of property *set*, additional $\delta$ operators may be introduced or removed without changing the query semantics. Rewrite 62 takes this information into account and reduces the amount of duplicate elimination operators.

$$\frac{\mathit{set}}{\delta(q) \rightarrow q} \tag{62}$$

$$\frac{\neg\mathit{set} \qquad \circledast \notin \{\delta, \Cup\} \qquad k \in \mathit{key} \qquad k \subseteq \mathit{icols}}{\circledast(q) \rightarrow \delta\left(\pi_{\mathit{icols}}(\circledast(q))\right)} \tag{63}$$



**(a)** Query plan of Query $Q_3$ after house cleaning and order rewrites.

**(b)** Query plan of (a) after application of Rewrites 47 and 52.

**(c)** Query plan of (b) after application of Rewrites 47 and 28, and `cse` simplification.

Figure 4.5: The effect of equi-join pushdown.

With Rewrite 62 in place, we now completely imitate *Rule 3. Distinct Pushdown From/To* of [99]. In combination with Rewrite 55, we effectively push distinct operators up in the plan and thus extend their scope with respect to the *set* information. Minimizing the number of $\delta$ operators, however, requires distinct operators to be pushed up actively.

Rewrite 63 implements the distinct operator pushup. The main idea of Rewrite 63 is to place a single $\delta$ near the plan root thus allowing all other distinct operators to be pruned. As this rewrite is neutralized by Rewrite 11 these two rewrites may never be used in the same rewrite class. A rewrite phase with Rewrites 62 and 63 active will ensure that a small number of distinct operators is placed nearest to the plan root. A subsequent application of Rewrite 11 may then remove the remaining, now superfluous, $\delta$ operators.

### 4.4.3 Dependency Disentanglement

For queries in the XQUERY workhorse dialect defined in [54]—that is, queries without positional variables, aggregates, and sequence construction—the rewrites proposed so far suffice to generate SQL queries that database systems are able to evaluate efficiently. Query $Q_6$ is such a workhorse query that gets rewritten into the query plan shown in Figure 4.6(a).

```
for $a in doc ("dblp.xml")/dblp/*
where doc ("dblp.xml")/dblp/*[title = "Join Graph Isolation"]
      /author = $a/author
return $a/title
```
$(Q_6)$

The plan features a single numbering operator and duplication elimination in the plan tail. The plan is turned into a single `SELECT DISTINCT·FROM·WHERE·ORDER BY` expression by the greedy SQL code generator. All conditions end up in a single conjunctive `WHERE` clause and the back-end may decide the evaluation order based on its statistics.

With respect to MonetDB/XQuery, the operator order in Figure 4.6(a) however is prescribed and follows the nested-loop semantics of the original query text: Before the first selection is performed, for example, a quadratic intermediate result is constructed.

The rewrites proposed in this section disentangle the dependencies leading to query plans in which independent plan fragments are split into separate subplans. As a side effect of the query unnesting, more sharing opportunities may be detected. In Figure 4.6(a), for example, two almost identical sets of operators implement the path `doc ("dblp.xml")/dblp/*`.

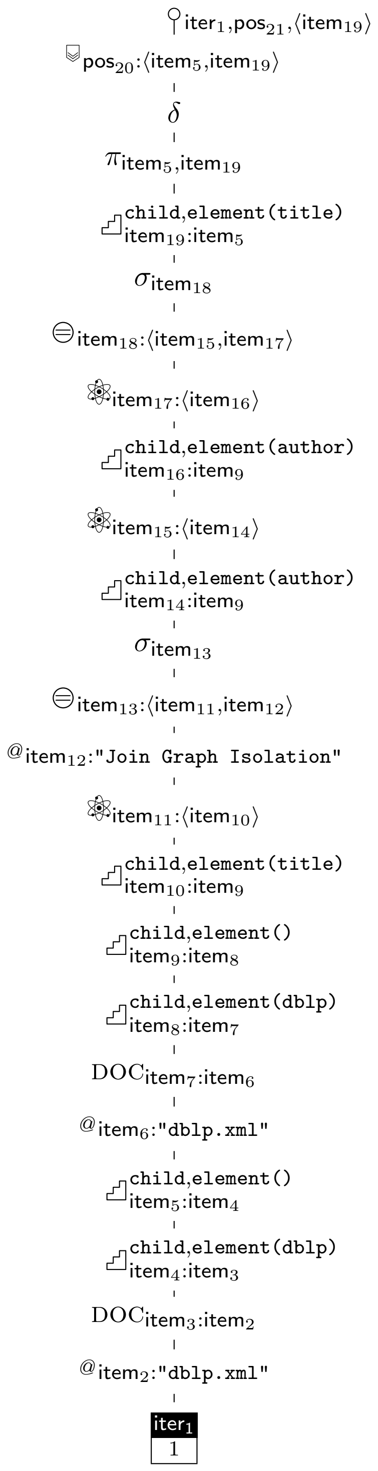
The basic idea of the dependency disentanglement is to push up cross products and joins. Rewrites 64 to 78 implement these transformations. As in Section 4.4.1, $p$ represents an arbitrary argument, $\Rrightarrow$ describes the operator reachability, and the dual rewrite rules are omitted.

$$\frac{\circledast \in \{\times, \bowtie_p\} \qquad cols(p) \subseteq \{a_1, \ldots, a_n, b_1, \ldots, b_m\} \\ \{a_1, \ldots, a_n\} \subseteq q_1.cols \qquad \{b_1, \ldots, b_m\} \subseteq q_2.cols}{\pi_{a_1,\ldots,a_n,b_1,\ldots,b_m}(q_1 \circledast q_2) \rightarrow (\pi_{a_1,\ldots,a_n}(q_1)) \circledast (\pi_{b_1,\ldots,b_m}(q_2))} (64)$$

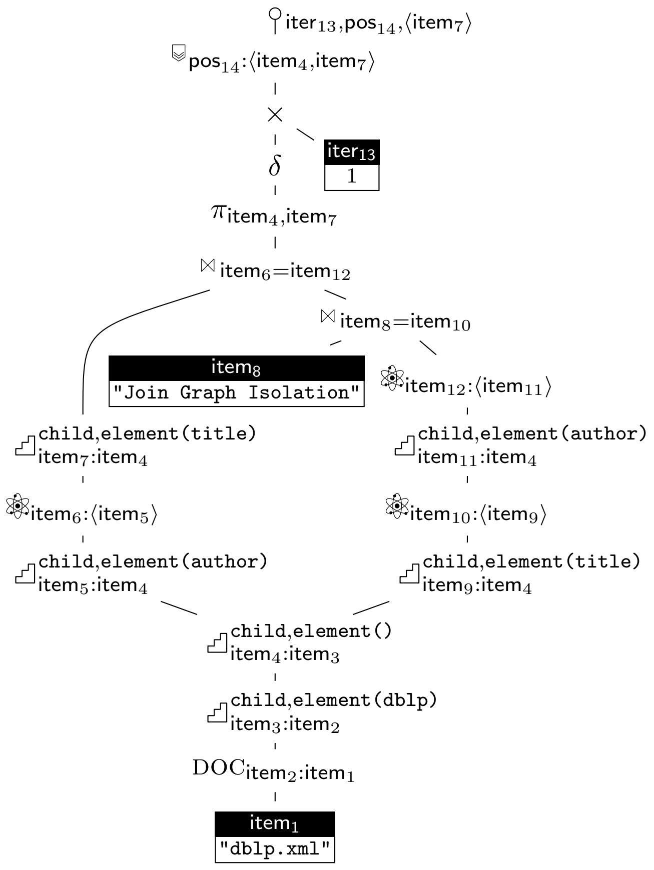
$$\frac{\circledast \in \{\times, \bowtie_p\} \qquad b \in q_1.cols}{\sigma_b(q_1 \circledast q_2) \rightarrow (\sigma_b(q_1)) \circledast (q_2)} (65) \qquad \frac{\circledast \in \{\times, \bowtie_p\}}{(q_1 \circledast q_2) \times q_3 \rightarrow q_1 \circledast (q_2 \times q_3)} (66)$$

$$\frac{\circledast \in \{\times, \bowtie_p\} \qquad a \in q_2.cols}{(q_1 \circledast q_2) \overset{=}{\bowtie}_{a=b} q_3 \rightarrow q_1 \circledast (q_2 \overset{=}{\bowtie}_{a=b} q_3)} (67)$$

$$\frac{\{a_{1\cdot 1}, \ldots, a_{1\cdot n}\} \subseteq q_1.cols \qquad \{b_{1\cdot 1}, \ldots, b_{1\cdot m}\} \subseteq q_2.cols}{\begin{array}{c}(q_1 \times q_2) \bowtie_{a_{1\cdot 1}\theta_1 a_{2\cdot 1} \wedge\cdots\wedge a_{1\cdot n}\theta_n a_{2\cdot n} \wedge b_{1\cdot 1}\theta_1 b_{2\cdot 1} \wedge\cdots\wedge b_{1\cdot m}\theta_m b_{2\cdot m}} q_3 \rightarrow \\ q_1 \bowtie_{a_{1\cdot 1}\theta_1 a_{2\cdot 1} \wedge\cdots\wedge a_{1\cdot n}\theta_n a_{2\cdot n}} (q_2 \bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1} \wedge\cdots\wedge b_{1\cdot m}\theta_m b_{2\cdot m}} q_3)\end{array}} (68)$$


iter1,pos21,⟨item19⟩
pos20:⟨item5,item19⟩
δ
π item5,item19
child,element(title) item19:item5
σ item18
item18:⟨item15,item17⟩
item17:⟨item16⟩
child,element(author) item16:item9
item15:⟨item14⟩
child,element(author) item14:item9
σ item13
item13:⟨item11,item12⟩
@ item12:"Join Graph Isolation"
item11:⟨item10⟩
child,element(title) item10:item9
child,element() item9:item8
child,element(dblp) item8:item7
DOC item7:item6
@ item6:"dblp.xml"
child,element() item5:item4
child,element(dblp) item4:item3
DOC item3:item2
@ item2:"dblp.xml"
iter1
1

**(a)** Query plan of Query $Q_6$.


iter13,pos14,⟨item7⟩
pos14:⟨item4,item7⟩
×
iter13
1
δ
π item4,item7
⋈ item6=item12
⋈ item8=item10
item8
"Join Graph Isolation"
item12:⟨item11⟩
child,element(title) item7:item4
child,element(author) item11:item4
item6:⟨item5⟩
item10:⟨item9⟩
child,element(author) item5:item4
child,element(title) item9:item4
child,element() item4:item3
child,element(dblp) item3:item2
DOC item2:item1
item1
"dblp.xml"

**(b)** Query plan of Query $Q_6$ after query unnesting.

Figure 4.6: The effect of dependency disentanglement.

$$\frac{\{a_{1\cdot 1},\ldots,a_{1\cdot n}\} \subseteq q_1.cols \qquad \{b_{1\cdot 1},\ldots,b_{1\cdot m}\} \subseteq q_2.cols}{\begin{array}{c}(q_1 \bowtie_p q_2) \bowtie_{a_{1\cdot 1}\theta_1 a_{2\cdot 1}\wedge\cdots\wedge a_{1\cdot n}\theta_n a_{2\cdot n}\wedge b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot m}\theta_m b_{2\cdot m}} q_3 \rightarrow \\ q_1 \bowtie_{p\wedge a_{1\cdot 1}\theta_1 a_{2\cdot 1}\wedge\cdots\wedge a_{1\cdot n}\theta_n a_{2\cdot n}} (q_2 \bowtie_{b_{1\cdot 1}\theta_1 b_{2\cdot 1}\wedge\cdots\wedge b_{1\cdot m}\theta_m b_{2\cdot m}} q_3)\end{array}} \quad (69)$$

$$\frac{\circledast \in \{\times, \bowtie_p\} \qquad q_1 \Rrightarrow q_3 \qquad q_3 \Rrightarrow q_1}{(q_1 \circledast q_2) \mathbin{\dot\cup} (q_3 \circledast q_4) \rightarrow q_1 \circledast (q_2 \mathbin{\dot\cup} q_4)} \quad (70) \qquad \frac{\circledast \in \{\times, \bowtie_p\} \qquad q_1 \Rrightarrow q_3 \qquad q_3 \Rrightarrow q_1}{(q_1 \circledast q_2) \setminus (q_3 \circledast q_4) \rightarrow q_1 \circledast (q_2 \setminus q_4)} \quad (71)$$

$$\frac{\{b_1,\ldots,b_n\} \subseteq q_1.cols \qquad b_{grp} \in q_2.cols}{\begin{array}{c}\text{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}} (q_1 \times q_2) \rightarrow \\ \left(\pi_{a_1,\ldots,a_n} \left(\text{GRP}_{a_1:\circ_1(b_1),\ldots,a_n:\circ_n(b_n)/b_{grp}} \left(@_{b_{grp}:1}(q_1)\right)\right)\right) \times \left(\pi_{b_{grp}}(q_2)\right)\end{array}} \quad (72)$$

$$\frac{}{@_{a:val}(q) \rightarrow q \times \begin{array}{|c|}\hline a \\ \hline val \\ \hline\end{array}} \quad (73) \qquad \frac{\circledast \in \{\times, \bowtie_p\} \qquad \{b_1,\ldots,b_n\} \subseteq q_1.cols}{\circledR_{a:\langle b_1,\ldots,b_n\rangle}(q_1 \circledast q_2) \rightarrow \left(\circledR_{a:\langle b_1,\ldots,b_n\rangle}(q_1)\right) \circledast q_2} \quad (74)$$

$$\frac{\begin{array}{c}(a,v) \in req \qquad b_1 \in q_1.cols \qquad b_2 \in q_2.cols \\ (\circledast, v, \theta) \in \{(\circleddash, \text{‘}true, =), (\circleddash, \text{‘}false, \neq), (\otimes, \text{‘}true, >), (\otimes, \text{‘}false, \leq)\}\end{array}}{\circledast_{a:\langle b_1,b_2\rangle}(q_1 \times q_2) \rightarrow (@_{a:v}(q_1)) \bowtie_{b_1\theta b_2} q_2} \quad (75)$$

$$\frac{\begin{array}{c}(a,v) \in req \qquad b_1 \in q_1.cols \qquad b_2 \in q_2.cols \\ (\circledast, v, \theta) \in \{(\circleddash, \text{‘}true, =), (\circleddash, \text{‘}false, \neq), (\otimes, \text{‘}true, >), (\otimes, \text{‘}false, \leq)\}\end{array}}{\circledast_{a:\langle b_1,b_2\rangle}(q_1 \bowtie_p q_2) \rightarrow (@_{a:v}(q_1)) \bowtie_{p\wedge b_1\theta b_2} q_2} \quad (76)$$

$$\frac{\{b_1,\ldots,b_n\} \subseteq q_1.cols \qquad b_{grp} \in q_2.cols}{\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle/b_{grp}}(q_1 \times q_2) \rightarrow \left(\vec{\#}_{a:\langle b_1,\ldots,b_n\rangle}(q_1)\right) \times q_2} \quad (77)$$

$$\frac{\circledast \in \{\times, \bowtie_p\} \qquad b \in q_1.cols}{⏌^{\alpha,v}_{a:\langle b\rangle}(q_1 \circledast q_2) \rightarrow \left(⏌^{\alpha,v}_{a:\langle b\rangle}(q_1)\right) \circledast q_2} \quad (78)$$

Similar to the equi-join pushdown rewrites, cross products and joins are pushed through their own kind in Rewrites 66 to 69. To guarantee the completion of the dependency disentanglement phase, the rewrites are scheduled in rounds where the operator closest to the plan root is pushed up first. If this operator cannot be pushed up further it is suspended til the next, and the next cross product or join is pushed up. The phase completes once Rewrites 66 to 69 were the only applicable rules in a round.

While the SQL code generator is able to overcome the operator nesting for many queries—as sketched in Figure 4.6(a)—the SQL-consuming database systems also benefit from the Rewrites 70 to 72 and the Rewrite 77, which decrease the input cardinality of the $\dot\cup$, $\setminus$, GRP, and $\vec{\#}$ operators.

More unnesting potential is introduced by Rewrite 73, which transforms an $@$ operator into a cross product. Rewrites 75 and 76 collect additional join predicates, if a comparison whose result is subject to a selection in the downstream plan ($req$), references columns from both inputs $q_1$ and $q_2$.[14]

[14]To mark the selection operator in the downstream plan for deletion, Rewrites 75 and 76 replace the result of the comparison by constant values.

The query plan in Figure 4.6(a) is subject to Rewrites 65, 73, 74, 75, and 78. Rewrite 73 turns the three @ operators into three independent literal tables. The subsequent rewrites effectively push down the ⊿, ⚛, and DOC operators into these independent branches. Rewrite 75 turns two of the three cross products into joins. Figure 4.6(b) shows the resulting plan where—in addition to the unnesting rewrites—house cleaning and algorithm `cse` have been applied. The common subplan elimination was able to eliminate one instance of the common path `doc("dblp.xml")/dblp/*`.

### 4.4.4 Related Work

The equi-join pushdown rewrites presented in Section 4.4.1 are (closely) related to the removal of correlations in nested subqueries. In other query compilers [43, 96], nested subqueries are represented by a *dependent join* operator [29, 46]. From the dependent join operator *Apply* ($\mathcal{A}^{\times}$) in [46] we adopt its definition:

$$q_1 \, \mathcal{A}^{\times}_{q_{bind}} \, q_2 = \bigcup_{r \in q_1} \left( \{r\} \times q_2\left[{}^{\{r\}}/_{q_{bind}}\right] \right) ,$$

where $q[{}^{x}/_{y}]$ denotes the consistent replacement of free occurrences of $y$ in $q$ by $x$. This definition mirrors the semantics of the `for` loop construct: $q_2$ (the loop body) is treated like a function with parameter $q_{bind}$ that is iteratively evaluated for each row $r$ of table $q_1$.

Rule FOR′ demonstrates how the loop lifted compiler may be aligned with the approaches just mentioned. Instead of using numbering operators and equi-joins to provide the loop information and align the result of the nested subquery with the outer query, respectively, the dependent join operator expresses the nested evaluation semantics ($q_{return \cdot apply}$).

$$\frac{\begin{array}{l} \Gamma; loop \vdash e_{in} \Mapsto q_{in} \\ q_v \equiv @_{\mathsf{iter:1}} \left( @_{\mathsf{pos:1}} \left( \pi_{\mathsf{item}} \left( q_{bind} \right) \right) \right) \\ \Gamma + \{\$v \mapsto q_v\} \, ; \pi_{\mathsf{iter}} \left( q_v \right) \vdash e_{return} \Mapsto q_{return} \\ q_{return \cdot apply} \equiv q_{in} \, \mathcal{A}^{\times}_{q_{bind}} \left( \pi_{\mathsf{pos_2:pos,item_2:item}} \left( q_{return} \right) \right) \\ q \equiv \pi_{\mathsf{iter,pos:pos}_{new}\mathsf{,item:item_2}} \left( \varrho_{\mathsf{pos}_{new}:\langle \mathsf{pos,pos_2} \rangle} \left( q_{return \cdot apply} \right) \right) \end{array}}{\Gamma; loop \vdash \texttt{for \$v in } e_{in} \texttt{ return } e_{return} \Mapsto q} (\textrm{FOR}')$$

As the evaluation of dependent joins is based on a nested-loop paradigm, [46] proposes a set of rewrite rules that push down the dependent joins. A dependent join is pushed down into its right-hand subtree until the right child contains no more references to the free row variables in $q_{bind}$. The rules in [46, Figure 4] are directly comparable to the rewrites in Section 4.4.1. *Identity 1* that replaces the dependent join by a cross product, for example, corresponds to Rewrites 46 and 47. The only difference can be observed in *Identity 7* (as opposed to Rewrite 54) where a dependent join is pushed through both inputs of a cross product. Our equi-join pushdown detects where the iteration references originate and thus provides a more well-informed rewrite by pushing down the equi-join into a single input only.

As mentioned in Section 4.4.2, [99] removes superfluous distinct operators by taking downstream distinct operators into account. We share these ideas also with [41], which removes distinct operators based on the analysis of XQuery Core queries.

The dependency disentanglement in Section 4.4.3 applies peephole-style rewrites based only on the required value property. [51] proposes an alternative variant, based on degenerated multi-valued dependencies [15, 40], that allows a more holistic rewrite to unnest a complete subplan. By pushing up cross products and joins, however, we maintain the degenerated multi-valued dependencies in each rewrite. Ultimately, this leads to similar query plans.

The query unnesting that results from the rewrites in Section 4.4.3 turns long chains of operators into bushy join graphs where more common subplans can be detected by algorithm `cse`. In consequence, the intermediate result size can decrease significantly.

Thanks to its materialization of intermediate results, MonetDB/XQuery benefits from both the unnested query plans and the common subplan elimination. The query plan in Figure 4.6(b), for example, runs orders of magnitudes faster in MonetDB/XQuery than its equivalent in Figure 4.6(a). The SQL code generator, on the other hand, hides the bushy nature of the resulting query plans. In many cases, the SQL back-ends, however, may still benefit from the query unnesting as the remaining numbering operators consume less rows.

## 4.5 Improving XML Queries

The algebra introduced in Section 2.2 comes with a number of XML-specific placeholder operators. Except for the key and functional dependency property inference, we however ignored their XML characteristics in the last sections. Additional XML-related properties and rewrites might significantly enhance the query plan optimization, for example, for XQuery queries. In this section we briefly sketch rewrites integrated in Pathfinder as well as possible extensions that might further improve the generated query plans.

Rewrite 79 is one of the most effective simplifications in Pathfinder. The rewrite turns the algebraic core of the XQuery shorthand `//`[15] into a descendant path step join. This rewrite becomes applicable only after the order information is removed.

$$\frac{b \notin \mathit{icols}}{\text{⎍}^{\texttt{child},v}_{a:\langle b\rangle}\left(\text{⎍}^{\texttt{descendant-or-self},\texttt{node()}}_{b:\langle c\rangle}(q)\right) \rightarrow \text{⎍}^{\texttt{descendant},v}_{a:\langle c\rangle}(q)} \tag{79}$$

[15]In XQuery, `//` is expanded into `/descendant-or-self::node()/` [16].

A query consisting of a chain of `child` path steps and a final `descendant` path step turns into a sequence of $\lrcorner$ operators and a distinct operator on the top. The query *ctx*`/a/b/descendant::c`, for example, turns into the query plan

$$\delta\left(\pi_c\left(\lrcorner^{\texttt{descendant},c}_{c:\langle b\rangle}\left(\lrcorner^{\texttt{child},b}_{b:\langle a\rangle}\left(\lrcorner^{\texttt{child},a}_{a:\langle ctx\rangle}(\ldots)\right)\right)\right)\right) \quad ,$$

where the $\delta$ operator may be removed as the context nodes of the `descendant` step are located in non-overlapping parts of the XML document. In Pathfinder, we deploy an additional *level* property that records their statically inferred level—the length of a path leading from a node to its document root—for columns of type node (if applicable). The level property is maintained for path step joins along the non-recursive axes `child`, `attribute`, `parent`, `self`, `following-sibling`, and `preceding-sibling` and removed for all other axes.

The level information extends the key and functional dependency property inference for path step joins and thus indirectly leads to the additional removal of distinct operators:

$$\begin{array}{lll}
\lrcorner^{\alpha,v}_{a:\langle b\rangle}(q) & \mathit{key} \leftarrow \ldots \cup \{\{a\} \cup (k \setminus \{b\}) \,|\, (b:v) \in q.\mathit{level}, & \\
& \qquad\qquad\qquad\qquad k \in q.\mathit{key}, k \cap \{b\} \neq \varnothing, & \\
& \qquad\qquad\qquad\qquad \alpha \in \{\texttt{descendant}, & \\
& \qquad\qquad\qquad\qquad\quad \texttt{descendant-or-self}\}\} & \\
\lrcorner^{\alpha,v}_{a:\langle b\rangle}(q) & \mathit{fd} \;\;\leftarrow \ldots \cup \{a \to b \,|\, (b:v) \in q.\mathit{level}, & \\
& \qquad\qquad \alpha \in \{\texttt{descendant}, \texttt{descendant-or-self}\}\} &
\end{array}$$

Pathfinder supports XML node construction by means of additional placeholder operators. The compilation rules that integrate the node constructors are described in [120]. We ignored these operators so far as they play only a minor role during optimization. Nevertheless, node constructors may become a performance bottleneck, because of their subtree copy semantics.

By generating query plans for twig constructors instead of single node constructors, we limit the construction overhead. Pathfinder furthermore collects usage information for columns of type node. This additional node usage property is used in the physical plan generation for MonetDB/XQuery to decide whether subtree copies may be replaced by references.

A heuristic that is currently not implemented in Pathfinder is the pushdown of $\lrcorner$ operators operating on constructed transient nodes. In certain scenarios, the node constructors and the path step joins operating on them could cancel each other out. Such a set of rewrites could lead to XQuery queries that effectively operate on a sequence of tuples instead of a sequences of generated nodes.

In [76], a *fusion* of XPath steps and node constructors is introduced. The fusion rewrites operate on the XQuery level and take special care to correctly handle document order and node identity. They, however, fail to detect tuples, as

XQuery operates on sequences of items only. The approach we sketched above, in contrast, focuses on the collapse of ⊿ operators and node constructors, whose newly generated document order and node identity is immaterial, by taking into account the *icols* and *use* properties.

Specific knowledge about XML document structure may be integrated with the optimization in terms of DataGuides [47]. This can unlock additional optimization potential. Pathfinder can take an extended variant of such path summaries into account that additionally records the minimum and maximum number of node occurrences for every guide child node. Pathfinder uses this information to improve the already existing properties (*key*, *fd*, *level*, etc.).

The DataGuide information may also be used to replace a chain of ⊿ operators by a single, new operator ◢ that performs a guide index join and thus avoids the evaluation of a potentially large number of path step joins. Such a rewrite would, however, rely on a back-end and an algebra operator ◢ that both encode the DataGuide information. As our objective is to keep the query compiler independent of the back-end and, more importantly, independent of a particular XML encoding, we let this chance to improve the query plans pass.

## 4.6 Summary

In Chapter 3 we identified (a) the DAG shape of the query plans, (b) the large number of non-traditional numbering operators, and (c) the large number of mapping joins as the main obstacles any query optimizer has to overcome to provide an efficient evaluation for loop lifted queries.

In this chapter we proposed three heuristics consisting of peephole-style rewrite rules taking the characteristics of the loop lifted algebra plans into account.

- **House Cleaning**. These rewrites prune a large number of superfluous operators and clean up leftovers of the other heuristics.
- **Order Minimization**. These rewrites minimize the amount of numbering operators that encode order information. The remaining numbering operators are necessary to either provide the overall output order or to implement a query's access to the sequence position.
- **Query Unnesting**. These rewrites resolve the mapping joins and disentangle dependencies. These rewrites also remove the references to most of the remaining numbering operators that encoded the iteration values. Subsequent house cleaning prune the unreferenced operators.

Once these optimizations are applied, for many queries traces of the loop lifted compilation strategy—clearly present in the initial plans—are no longer recognizable. Similarly, syntactic variants of the input programs are indistinguishable after the rewrites. The generated SQL code for many queries resembles the preeminent `SELECT·FROM·WHERE` queries most database back-ends are geared to support efficiently. The remaining plan sharing is handled by the back-end: The sharing might be resolved by either replication [99], a DAG-aware query processor [90], or a back-end with plan recyclers [70].

For back-ends that evaluate the resulting plans more or less unmodified, such as MonetDB/XQuery, query unnesting plays an important role. The optimizations proposed here, however, provide only amelioration—no optimal plan. The rewrites, for example, do not prescribe the order of the two joins in Figure 4.6(b). Similarly, the evaluation order of the `author` and `title` path step joins may be changed.

Physical query optimization [26, 48, 72], on the other hand, takes data characteristics into account and may provide an optimal execution plan. The complete body of work on physical query optimization is orthogonal to our approach. *Any* database back-end may further improve the query plans.

In the context of MonetDB/XQuery, groups of operators (without numbering, distinct, aggregate, union, and difference operators) form join graphs that are subject to runtime query optimizations [74]. Sampling and *zero-investment* algorithms detect data correlation and decide for the most efficient evaluation order at runtime. This approach further improves join tree planning and overcomes selectivity misestimation issues of classical optimizers. While this runtime optimizer is not yet integrated with MonetDB/XQuery, in a future version it might provide an optimal execution plan for the query plan in Figure 4.6(b).

# Chapter 5

# Optimization Assessment

The previous chapter provided a glimpse of the optimizations' impact. Here, we examine the effects of the rewrite process in more detail. We analyze the query shape of the resulting query plans as well as the query plan stability with respect to different query formulations (Section 5.1). Section 5.2 gives an impression on how the back-ends MonetDB/XQuery and DB2 react to the optimized workloads. We analyze the performance of XQuery queries on MonetDB/XQuery and let DB2 perform optimizations on the rewritten plans. A performance comparison of the TPC-H benchmark queries [121] with their loop lifted counterparts finally underlines the robustness of the rewrites with respect to evaluation time.

## 5.1 Analyzing the Query Plan Shape

As the optimizations are independent of any back-end query processor, in the following we focus on the quality of the resulting query plans. In Chapter 3 we analyzed the operator distribution for the XMark benchmark query set. Table 5.1 replicates Table 3.1 and extends the table with the operator counts and their distribution after Pathfinder has optimized the initial loop lifted query plans.

In comparison to the initial loop lifted query plans, the average rewritten query plan features a sixth of the operators (43.50 instead of 262.65 operators), of which over 50 percent are XML operators. The rewrites eliminated more than 95 percent of the numbering operators, mapping joins, and distinct operators and furthermore detected 6 value-based joins in XMark Queries 8 to 12.

Most of the 20 XMark query plans consist of one or more join graph bundles—operators that can be turned into a single `SELECT·FROM·WHERE` expression—and a subsequent plan tail that performs, duplicate elimination, aggregation and node construction. Figure 5.1 shows the rewritten query plan for XMark Query 8. The optimizations led to a significantly smaller plan—31 instead of the initial 286 operators—consisting of four join graph bundles (below the dotted line), which perform most of the work, and a plan tail with duplicate elimination, aggregation,

| **Operator Category** | **Loop Lifted Query Plans** | | **Optimized Query Plans** | |
|---|---|---|---|---|
| | Average # of Operators | Fraction of Operators | Average # of Operators | Fraction of Operators |
| Projections | 130.65 | 49.7% | 7.20 | 16.6% |
| Row Operators | 39.55 | 15.1% | 7.90 | 18.2% |
| Numbering Operators | 35.25 | 13.4% | 0.45 | 1.0% |
| XML Operators | 25.70 | 9.8% | 22.40 | 51.5% |
| Mapping Joins | 14.45 | 5.5% | 0.45 | 1.0% |
| Duplicate Elimination | 6.75 | 2.6% | 0.60 | 1.4% |
| Set Operators | 6.40 | 2.4% | 1.30 | 3.0% |
| Value-Based Joins | 0.00 | 0.0% | 0.30 | 0.7% |
| Others | 3.90 | 1.5% | 2.90 | 6.7% |
| Overall | 262.65 | 100.0% | 43.50 | 100.0% |

Table 5.1: Categorized operator distribution for the average XMark query before and after optimization.

and constructors (above the dotted line in Figure 5.1). This clear separation is characteristic for most XMark queries.

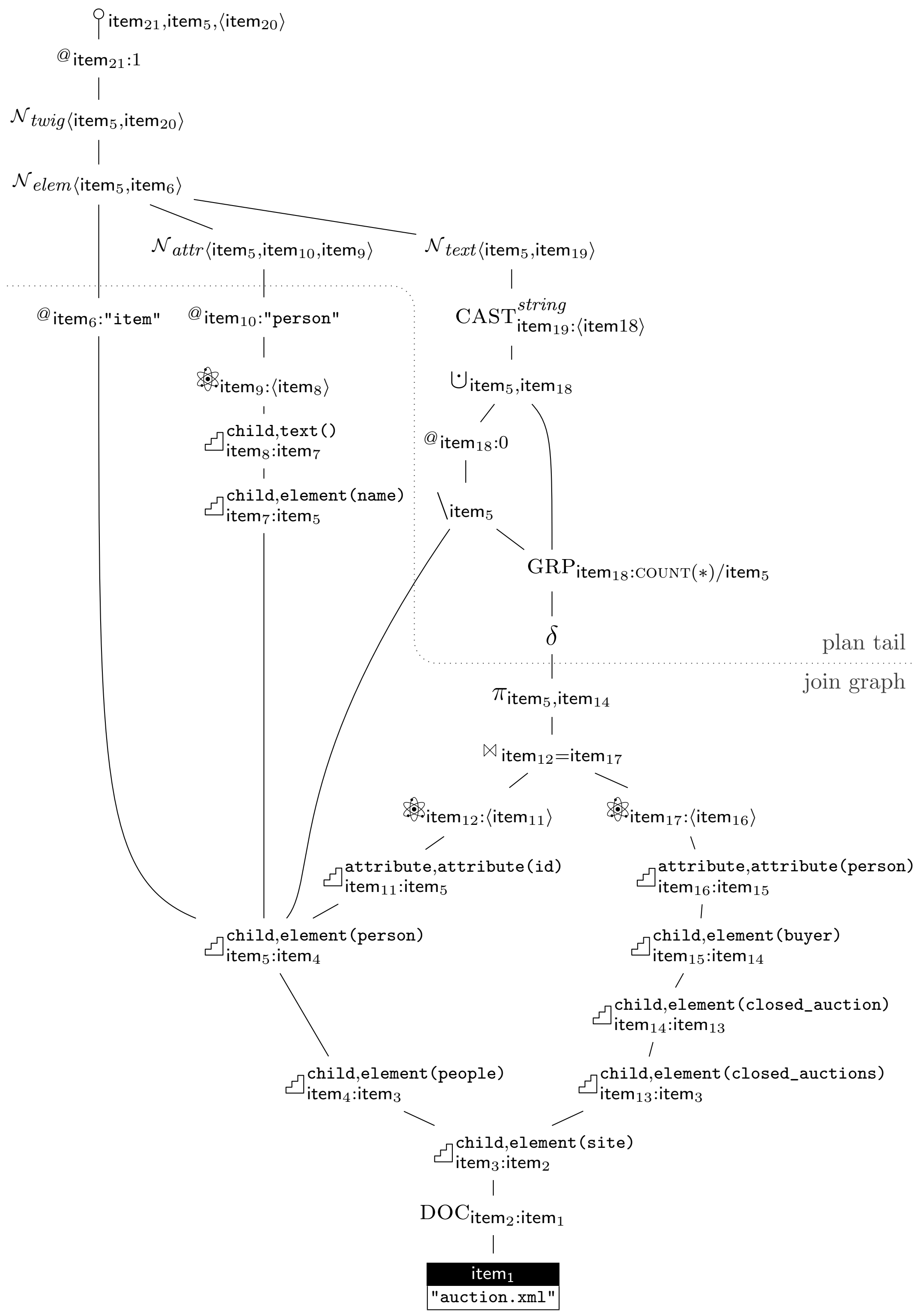
An assessment of the rewrites' effect on the evaluation times of the XMark queries follows in Section 5.2.1. First, however, we examine the robustness of the optimizations with respect to the initial loop lifted query plans for two further query scenarios.

## 5.1.1 Detection of Value-Based Joins

In [2], Afanasiev describes a micro-benchmark for value-based equi-joins. This benchmark is used to showcase the MemBeR micro-benchmarking methodology [6] and the XCheck benchmark platform [3]. The benchmark provides a comprehensive view on the performance of various query processors on value-based joins without looking into the engines' internals. One of the systems evaluated in [2] is MonetDB/XQuery.

As the experiments conducted with MonetDB/XQuery relied on the optimizations described in Chapter 4, we think of this micro-benchmark as especially interesting. Here, we delve into Pathfinder's internals and analyze the rewritten query plans. The observations collected during the examination allow us to better explain the performance characteristics reported in [2] and, more interestingly, provide an insight into the capabilities of the optimizations.

Afanasiev takes seven parameters into account: (a) syntactic pattern, (b) number of join conditions, (c) Boolean connectives, (d) join type, (e) join selectivity, and (f) join input size. Only the first three parameters lead to different input



item21,item5,⟨item20⟩
@item21:1
N twig⟨item5,item20⟩
N elem⟨item5,item6⟩
N attr⟨item5,item10,item9⟩
N text⟨item5,item19⟩
@item6:"item"
@item10:"person"
CAST string item19:⟨item18⟩
∪ item5,item18
item9:⟨item8⟩
child,text() item8:item7
@item18:0
child,element(name) item7:item5
\item5
GRP item18:COUNT(*)/item5
δ
plan tail
join graph
π item5,item14
⋈ item12=item17
item12:⟨item11⟩
item17:⟨item16⟩
attribute,attribute(id) item11:item5
attribute,attribute(person) item16:item15
child,element(person) item5:item4
child,element(buyer) item15:item14
child,element(closed_auction) item14:item13
child,element(people) item4:item3
child,element(closed_auctions) item13:item3
child,element(site) item3:item2
DOC item2:item1
item1
"auction.xml"

Figure 5.1: Rewritten algebraic query plan of XMark Query 8.

queries and affect the query plan analysis. In the following, we briefly summarize the specification of the three parameters.

In [2], Afanasiev discusses four different syntactic patterns to express a value-based equi-join. The following query skeletons sketch the different patterns:

```
(Where)      for $a in A, $b in B
             where $a/@att1 = $b/@att2
             return C
(Predicate)  for $a in A, $b in B[$a/@att1 = ./@att2]
             return C
(If)         for $a in A, $b in B
             return if ($a/@att1 = $b/@att2)
                    then C else ()
(Filter)     for $a in A, $b in B
             return C[$a/@att1 = $b/@att2] ,
```

where $A$ and $B$ represent independent path expressions[1] and $C$ is the sequence construction `($a/@att1,$b/@att2)`. Afanasiev furthermore makes use of five combinations of Boolean connectives and a variety of join conditions:

| | |
|---|---|
| ($Cond_1$) | a single join condition (as sketched in the skeletons above), |
| ($Cond_{2 \cdot \wedge}$) | two conjunction conditions, |
| ($Cond_{2 \cdot \vee}$) | two disjunctive conditions, |
| ($Cond_{3 \cdot \wedge \cdot \wedge}$) | three conjunctive conditions, and |
| ($Cond_{3 \cdot \wedge \cdot \vee}$) | two conjunctive conditions followed by a disjunctive condition. |

Based on these criteria we now repeat the analysis. We first inspect the four different syntactic patterns in combination with a single join condition ($Cond_1$) and a simple binding for $C$, namely `$b`. As `$a` does not appear free in $B$, the rewrites can always unnest $B$. All four syntactic patterns lead to the *same* rewritten query plan depicted in Figure 5.2. For these four queries we therefore expect identical evaluation times.

In [2], the syntactic pattern (*Filter*), however, shows inferior performance for MonetDB/XQuery. Figure 5.2 marks the regions of the query plan corresponding to the expressions $A$, $B$, and $C$ as well as the left and right join arguments ($L$ and $R$, respectively). We use this abstract notation to compare the four rewritten query plans for the original value of $C$—`($a/@att1,$b/@att2)`—used in [2]. Figure 5.3 shows the different plans in which $C$ represents the sequence construction. The patterns (*Where*), (*Predicate*), and (*If*) share the same query plan. Thanks to the greedy nature of the unnesting rewrites the plan in Figure 5.3(a) features two value-based joins. Pattern (*Filter*) in Figure 5.3(b) misses these joins as the cross

[1]In [2], the path expressions $A$ and $B$ are used to adjust the input selectivities of the joins.

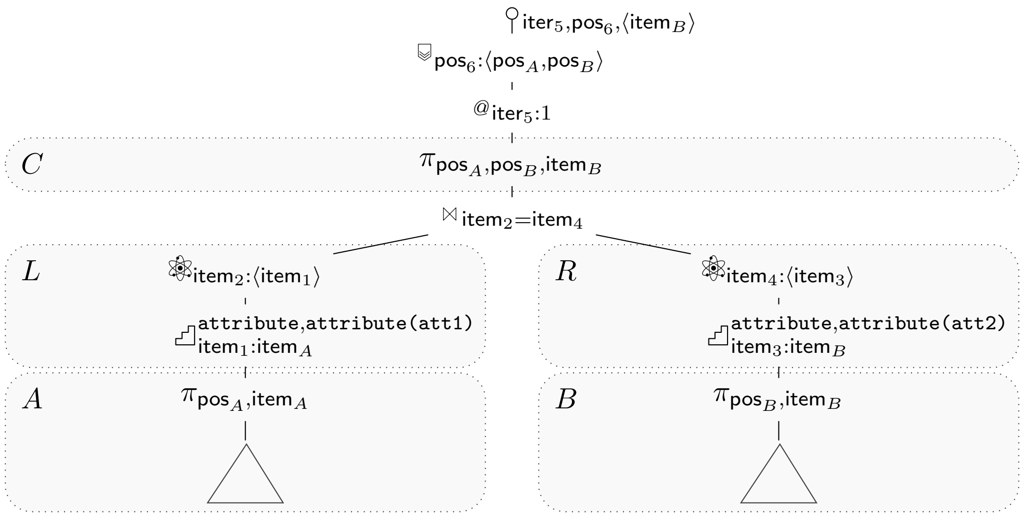
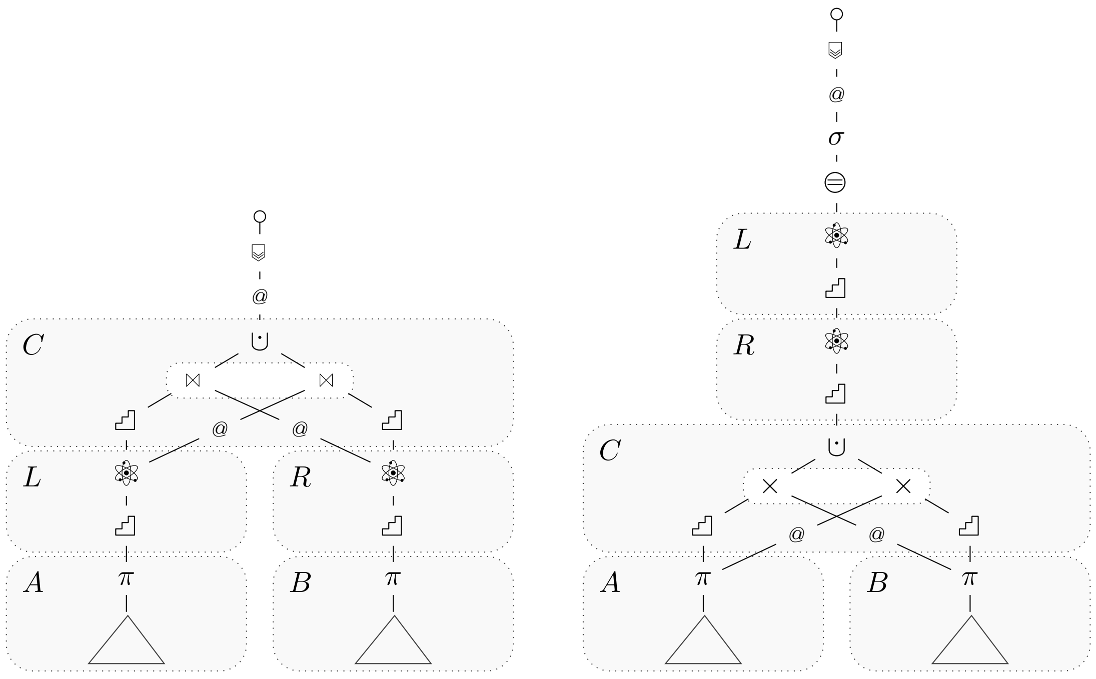


Figure 5.2: Rewritten query plan for all four syntactic patterns of the value-based equi-join query with a single comparison ($Cond_1$) and $C \equiv$ `$b`. (Plan fragments $L$ and $R$ depict the left and right arguments of the XQuery join condition.)



**(a)** Patterns (*Where*), (*Predicate*), (*If*). **(b)** Pattern (*Filter*).

Figure 5.3: Query shape of the rewritten query plans for the different syntactic patterns of the value-based equi-join query with a single comparison ($Cond_1$) and $C \equiv$ `($a/@att1,$b/@att2)`.

products that stem from the query unnesting of $B$ may not be pushed through the union operator implementing XQuery's sequence construction. With the current set of rewrite rules, pattern (*Filter*) leads to plans in which the plan fragment $C$ is located below $L$ and $R$.

This observation explains the evaluation times observed in [2]. We, however, also witnessed that (*Filter*) is not, a priori, the inferior alternative (Figure 5.2). For any XQuery query in the workhorse dialect, for example, Pathfinder detects the value-based joins independent of the syntactic pattern.

Interestingly, the two differing query plans in Figure 5.3 lead to the same execution plan in DB2. DB2 is able to push $L$, $R$ and the join condition down through the union operator. With a similar set of rewrites as employed by DB2, a future Pathfinder version could improve the robustness of MonetDB/XQuery with respect to syntactic patterns.

Because of Rewrite 17, we are able to resolve $\owedge$ operators that implement XQuery's `and` clause. The previous observations thus also apply for the join conditions ($\mathit{Cond}_{2\cdot\wedge}$) and ($\mathit{Cond}_{3\cdot\wedge\cdot\wedge}$). This reasoning is supported by the performance of MonetDB/XQuery measured in [2]. Our rule set, however, does not feature a rule that is able to split an $\ovee$ operator that implements XQuery's `or` clause. For the join conditions ($\mathit{Cond}_{2\cdot\vee}$) and ($\mathit{Cond}_{3\cdot\wedge\cdot\vee}$) the missing rule results in query plans in which Rewrites 75 and 76 are not applicable—that is, the cross products between $A$ and $B$ are not pushed up to form a join. Similar to the (*Filter*) pattern, these cross products lead to a performance drop of MonetDB/XQuery in [2], whereas DB2's optimizer is able to efficiently support these query plans.

This micro-benchmark demonstrates the robustness of the rewrites (Figure 5.2 and Figure 5.3(a)), but also shows some deficiencies (Figure 5.3(b) and conditions ($\mathit{Cond}_{2\cdot\vee}$) and ($\mathit{Cond}_{3\cdot\wedge\cdot\vee}$)) with respect to the code generation in MonetDB/XQuery. These shortcomings are no obstacle in general—the generated SQL code after simplification provides, for example, enough freedom for DB2 to further optimize the query plans.

### 5.1.2 Query Formulation

The second scenario operates on the wholesale supplier data from the TPC-H benchmark [121]. The query we analyze is based on the information need:

"List the order items of the order with the most parts." ($Q_7$)

We formulate Query $Q_7$ from two different perspectives: (a) a programmer, who takes ordered and complex data structures for granted, and (b) a database application developer, who tries to compose a query that can be efficiently evaluated by a relational database system.

The Linq program in Figure 5.4(a) shows a programmer's variant of Query $Q_7$. The program groups `lineitems` by `orderkey`, before ordering the groups by the sum of the quantities within each group. Taking the first group ensures that the

```
var q = db.LINEITEM
          .GroupBy(li => li.L_ORDERKEY)
          .OrderByDescending(g => g.Sum(li => li.L_QUANTITY))
          .ElementAt(0);
```

**(a)** Query written from a programmer's point of view (using group, nesting, and positional information).

```
var grps = from li in db.LINEITEM
           group li by li.L_ORDERKEY into g
           select new { ok = g.Key,
                        sum = g.Sum(li => li.L_QUANTITY)};

var q = from li in db.LINEITEM
        join grp in grps on li.L_ORDERKEY equals grp.ok
        where grp.sum == grps.Max(g => g.sum);
        select li;
```

**(b)** Query written from a database application developer's point of view (using group, self join, and maximum).

```
var grps = from li in db.LINEITEM
           group li by li.L_ORDERKEY into g
           select new { ok = g.Key,
                        sum = g.Sum(li => li.L_QUANTITY),
                        lis = g };

var q = from g in grps
        from li in g.lis
        where grp.sum == grps.Max(g => g.sum);
        select li;
```

**(c)** Query with concepts of (a) and (a) mixed (using group, nesting, and maximum).

```
var g = db.LINEITEM
          .Select(li => li.L_ORDERKEY)
          .Distinct()
          .Select(keys
          => new { ok = keys,
                   lis = db.LINEITEM
                          .Where(li
                          => li.L_ORDERKEY = keys) });
```

**(d)** Query snippet replacing LINQ's grouping construct.

Figure 5.4: LINQ variants of Query $Q_7$.

items of the order containing the most parts are returned. Within each group `g`, LINQ stores the grouping criteria (`g.Key`) as well as the list of corresponding records.[2]

Writing the query with a database background might lead to the LINQ program in Figure 5.4(b). This variant is written in LINQ's query syntax as opposed to the method syntax in Figure 5.4(a). The query uses grouping, a maximum aggregate, and a self join to retrieve the order items with the most parts. The query is based on a SQL formulation of the query and ignores LINQ's nested data model as well as the list order.

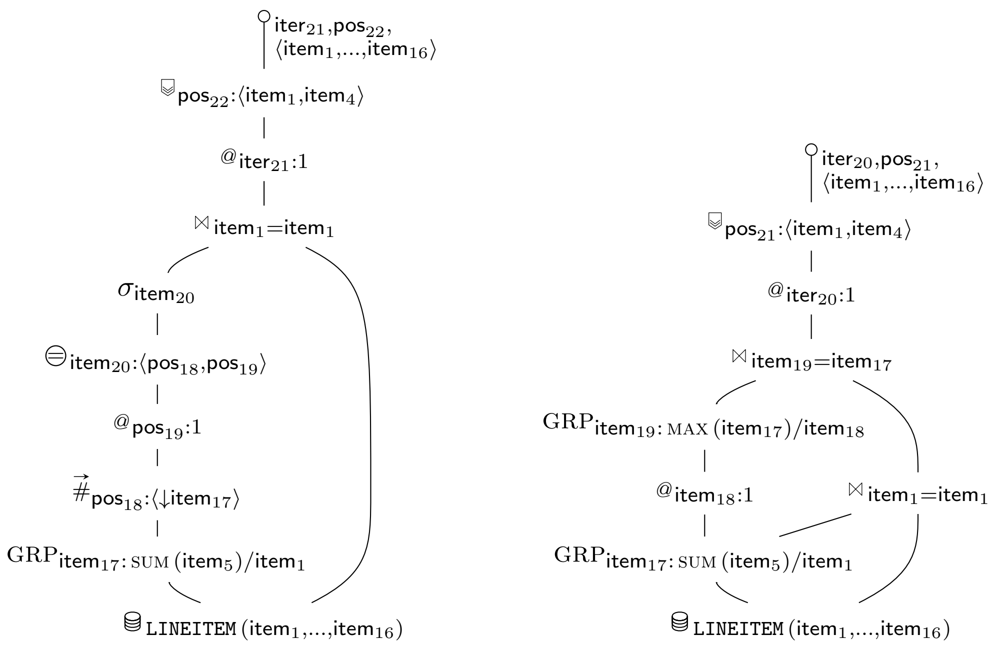
For both query variants, the optimizations significantly simplify the query plans, leading to almost minimal representations of the queries: Less than 10 operators encode the final plans (Figure 5.5). The query plans in Figure 5.5 differ, because of the different query formulations: Figure 5.4(a) returns the order items of a *single* order, whereas Figure 5.4(b) may return the order items of multiple orders—those with an identical number of parts. For varying TPC-H scale factors the queries provide the same result and the evaluation times on DB2 are almost identical for both query plans (with the position-based approach being slightly faster).

In addition to the difference of choosing the largest order, the queries in Figure 5.4(a) and Figure 5.4(b) differ in further query details. The query in Figure 5.4(a) uses an explicit self join, whereas the query in Figure 5.4(b) takes advantage of LINQ's nested data model. In the query plans this difference is not observable anymore: Both variants encode the relationship between the sums and the order items with an equi-join ($\bowtie_{\mathsf{item}_1 = \mathsf{item}_1}$). This observation is supported by a third variant of Query $Q_7$ (Figure 5.4(c)), which exploits LINQ's nested data model to avoid the explicit self join in Figure 5.4(b). The optimizations resolve the nesting and turn this query into the query plan in Figure 5.5(b).

Further transformations of the queries in Figure 5.4(a) and Figure 5.4(b) that replace the `group by` construct by a combination of an explicit duplicate elimination and a self join that retrieves the order items into a nested result

[2]We picked LINQ instead of LL as input language, to provide a clear view of the different query concepts. The following LL query is equivalent to the query in Figure 5.4(a). Its `for` loops and the positional (unnamed) access, however, interfere with the readability:

```
for $g at $p in for $g in group (table LINEITEM (16, (1,4)).1,
                                 table LINEITEM (16, (1,4)))
                order by sum (for $li in $g.2
                              return $li.5) descending
                return $g
return if ($p = 1)
       then $g.2
       else empty
```


iter21,pos22, ⟨item1,...,item16⟩
pos22:⟨item1,item4⟩
@iter21:1
⋈ item1=item1
σitem20
⊖item20:⟨pos18,pos19⟩
@pos19:1
#pos18:⟨↓item17⟩
GRPitem17: SUM (item5)/item1
LINEITEM (item1,...,item16)
iter20,pos21, ⟨item1,...,item16⟩
pos21:⟨item1,item4⟩
@iter20:1
⋈ item19=item17
GRPitem19: MAX (item17)/item18
@item18:1
⋈ item1=item1
GRPitem17: SUM (item5)/item1
LINEITEM (item1,...,item16)

**(a)** Rewritten position-based query plan of the query in Figure 5.4(a).

**(b)** Rewritten aggregate-based query plan of the query in Figure 5.4(b).

Figure 5.5: Rewritten query plans of Query $Q_7$.

(Figure 5.4(d)), still result in the rewritten query plans in Figure 5.5.

The most compact variant of Query $Q_7$, we came up with, is written in Ruby:

```
LINEITEM.group_by{ |x| x.L_ORDERKEY }
         .max_by{ |ok,lis| lis.sum { |li| li.QUANTITY } }
```

The `max_by` construct calculates its body expression, selects the single, maximal value, and returns the corresponding argument `|ok,lis|`—the order items of the order with the most parts. The query plan resulting from this Ruby query hence mixes the concepts of both query plans in Figure 5.5 by first performing a filter on the maximum, before applying the positional predicate.

## 5.2 Quantitative Evaluation

In the following, we turn our focus to the runtime characteristics of the optimized query plans. We demonstrate the runtime effect of the simplifications, examine how DB2 copes with a rewritten XQuery workload, and assess the overhead of expressing SQL queries in a non-relational language.

All experiments were conducted on the same host—a Sun Fire X4275 server. The host is equipped with two Intel Xeon processors X5570 (2.93 GHz; quad core),

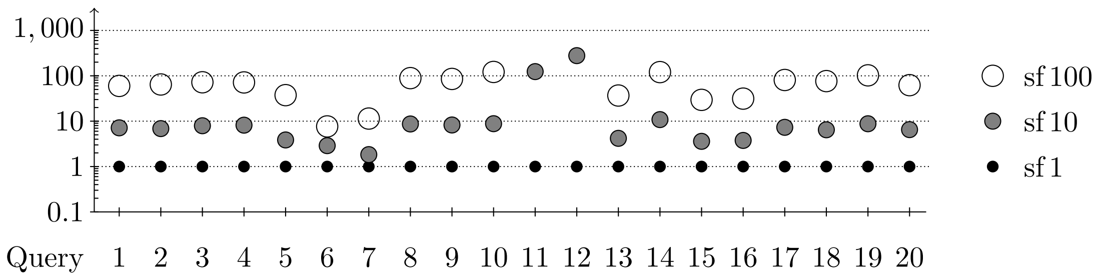


Figure 5.6: Normalized evaluation times of the optimized XMark queries, which emphasize the scalability of MonetDB/XQuery with respect to the document size.

72 GB main-memory, and 6 TB of disk space. The queries were compiled and optimized with Pathfinder included in MonetDB/XQuery (v0.36.2). measurements were performed on an IBM DB2 database system (V9.7.0, fixpack 0) and a MonetDB Server (v4.36.2).

## 5.2.1 XMark on MonetDB/XQuery

For varying scale factors (sf), we collected the evaluation times of the 20 XMark queries both for the initial loop lifted query plans (sf 0.01–1) and the optimized query variants (sf 0.01–100). We ran each query 10 times and reported the average wall-clock evaluation time (in milliseconds). Table 5.2 lists the results of the measurements. No rewritten query performs worse than its initial loop lifted variant. In fact, the optimized variants of XMark Queries 6 to 12 halved the evaluation times (see $\overset{x}{>}$ in Table 5.2) even for the smallest document size (sf 0.01). The positive effect of the rewrites becomes even more evident for larger document instances. For scale factor 1, 75 percent of the queries are at least twice as fast as their initial loop lifted variants.

The big performance differences for XMark Queries 6 and 7 shown in Table 5.2 stem from the application of Rewrite 79 merging steps of the XQuery shorthand // (after the removal of ordering and duplicate elimination operators). XMark Queries 8 to 12 feature value-based joins, all of which are detected by the optimizations. In consequence, the rewritten queries run orders of magnitudes faster.

Figure 5.6 depicts the normalized evaluation times of the optimized query plans on MonetDB/XQuery (with respect to the evaluation times for scale factor 1). The graph shows that MonetDB/XQuery scales linearly with the document size. The only outliers are XMark Queries 11 and 12. In both queries, the bottleneck is a theta-join (comparison via >) that generates an intermediate result in the order of $10^8$ tuples for the scale factor 100 document. This concerns the query result, whose computation cannot be avoided (though the end result becomes small, thanks to subsequent aggregation).

| **XMark** | **sf 0.01** | | | **sf 0.1** | | | **sf 1** | | | **sf 10** | **sf 100** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Query** | initial | | opt | initial | | opt | initial | | opt | opt | opt |
| 1 | 19 | | 13 | 21 | | 15 | 98 | $\overset{2}{>}$ | 37 | 264 | 2,216 |
| 2 | 21 | | 16 | 25 | | 17 | 117 | $\overset{3}{>}$ | 36 | 247 | 2,320 |
| 3 | 42 | | 39 | 61 | | 37 | 271 | $\overset{2}{>}$ | 106 | 841 | 7,637 |
| 4 | 33 | | 23 | 39 | | 25 | 206 | $\overset{2}{>}$ | 78 | 637 | 5,578 |
| 5 | 19 | | 15 | 19 | | 15 | 41 | | 35 | 135 | 1,312 |
| 6 | 22 | $\overset{2}{>}$ | 10 | 97 | $\overset{10}{>}$ | 9 | 946 | $\overset{10}{>}$ | 9 | 26 | 69 |
| 7 | 41 | $\overset{3}{>}$ | 12 | 253 | $\overset{10}{>}$ | 11 | 2,482 | $\overset{10}{>}$ | 12 | 22 | 137 |
| 8 | 182 | $\overset{7}{>}$ | 24 | 11,870 | $\overset{10}{>}$ | 34 | 2,278,786 | $\overset{10}{>}$ | 161 | 1,397 | 14,322 |
| 9 | 204 | $\overset{5}{>}$ | 38 | 12,923 | $\overset{10}{>}$ | 47 | 2,286,729 | $\overset{10}{>}$ | 172 | 1,418 | 14,678 |
| 10 | 144 | $\overset{2}{>}$ | 68 | 1,359 | $\overset{10}{>}$ | 110 | 153,559 | $\overset{10}{>}$ | 588 | 5,176 | 71,528 |
| 11 | 143 | $\overset{4}{>}$ | 29 | 7,907 | $\overset{10}{>}$ | 43 | 1,509,956 | $\overset{10}{>}$ | 547 | 67,737 | DNF |
| 12 | 60 | $\overset{2}{>}$ | 26 | 2,400 | $\overset{10}{>}$ | 39 | 318,396 | $\overset{10}{>}$ | 310 | 86,310 | DNF |
| 13 | 23 | | 19 | 26 | | 18 | 66 | $\overset{2}{>}$ | 27 | 113 | 993 |
| 14 | 28 | | 18 | 131 | $\overset{2}{>}$ | 57 | 1,166 | $\overset{2}{>}$ | 431 | 4,669 | 52,030 |
| 15 | 29 | | 27 | 31 | | 25 | 40 | | 32 | 115 | 944 |
| 16 | 35 | | 31 | 36 | | 31 | 49 | | 38 | 143 | 1,194 |
| 17 | 20 | | 18 | 26 | | 21 | 117 | | 63 | 461 | 5,097 |
| 18 | 15 | | 13 | 17 | | 13 | 45 | | 26 | 168 | 1,988 |
| 19 | 32 | | 21 | 70 | $\overset{2}{>}$ | 31 | 496 | $\overset{3}{>}$ | 129 | 1,132 | 13,217 |
| 20 | 59 | $\overset{2}{>}$ | 21 | 68 | $\overset{2}{>}$ | 24 | 335 | $\overset{4}{>}$ | 74 | 482 | 4,607 |

Table 5.2: A comparison of MonetDB/XQuery's evaluation times for the queries of the XMark benchmark: initial loop lifted vs. optimized query plans (in msec, averaged over 10 runs). $\overset{x}{>}$ indicates that a query's optimized variant runs at least $x$ times faster.

| | **Pathfinder + DB2** | | | **MonetDB/XQuery** | | |
|---|---|---|---|---|---|---|
| | sf 0.01 | sf 0.1 | sf 1 | sf 0.01 | sf 0.1 | sf 1 |
| Loop Lifted Query Plans | 477 | 2,346,302 | – | 182 | 11,870 | 2,278,786 |
| Rewritten Query Plans | 63 | 1,982 | 62,227 | 24 | 34 | 161 |

Table 5.3: Evaluation times of XMark Query 8 (in msec, averaged over 10 runs).

In comparison to the best evaluation times of the XMark queries reported in [107], the current version of MonetDB/XQuery can almost keep up. On the same machine and back-end, the hand-optimized translation of XMark Query 8 (Figure 3.5) is only 2.6 times faster (62 msec).

**XMark on DB2.** In Section 3.3 we measured the evaluation times of the initial loop lifted query plan for XMark Query 8 both on MonetDB/XQuery and DB2. Table 5.3 extends Table 3.2 with the evaluation times of the rewritten query plans. Although the execution times for the rewritten query plans on DB2 are much faster, there is still a big performance gap compared to MonetDB/XQuery. One reason is the complex representation of node constructors on the SQL level. In [55], we demonstrated that the materialization of intermediate results in terms of temporary tables can overcome this performance bottleneck, but may pollute the database table space.

### 5.2.2 Join Graph Isolation on DB2

The initial loop lifted query plans forced DB2 to execute the queries in the given order: Because of the numbering operators, DB2 could not reorder any join operators. After applying our optimizations, DB2 is faced with query workloads for which its cost-based optimizer may start providing alternative efficient execution plans.

In the following, we analyze the result produced by DB2's optimizer—the execution plan—of an archetypical XQuery query formulated in the workhorse dialect [54]. Query $Q_8$ operates on XMark data and returns the names of those auction categories in which expensive items were sold (at prices beyond \$500):

```
let $a := doc("auction.xml")
for $ca in $a//closed_auction[price > 500],
    $i  in $a//item,
    $c  in $a//category                          (Q8)
where $ca/itemref/@item = $i/@id
  and $i/incategory/@category = $c/@id
return $c/name
```

Based on the XML encoding described in Section 3.2, the rewritten query plan of Query $Q_8$ is transformed into the SQL query of Figure 5.7. The query

```
 1  SELECT DISTINCT d13.*, d2.pre AS unq1, d4.pre AS unq2,
 2         d5.pre AS unq3, d12.pre as unq4,
 3    FROM DOC AS d1, ..., DOC AS d13
 4   WHERE d1.kind = DOC
 5     AND d1.name = 'auction.xml'
 6     AND d2.kind = ELEM
 7     AND d2.name = 'closed_auction'
 8     AND d2.pre BETWEEN (d1.pre + 1) AND (d1.pre + d1.size)
 9     AND d3.kind = ELEM
10     AND d3.name = 'price'
11     AND d3.pre BETWEEN (d2.pre + 1) AND (d2.pre + d2.size)
12     AND d2.level + 1 = d3.level
13     AND d3.data > 500
       ...
40     AND d12.kind = ELEM
41     AND d12.name = 'name'
42     AND d12.pre BETWEEN (d5.pre + 1) AND (d5.pre + d5.size)
43     AND d13.pre BETWEEN (d12.pre) AND (d12.pre + d12.size)
44   ORDER BY unq1, unq2, unq3, unq4, d13.pre
```

Figure 5.7: SQL encoding of Query $Q_8$.

describes a 12-fold self-join over table DOC. The structural node relationships of the step join operators $\lrcorner^{\alpha,v}$ expressed by $\alpha$ are mapped into conjunctive range join predicates over columns pre, size, and level: Line 8 in Figure 5.7 describes a step along the `descendant` axis and Lines 11–12 a `child` step. Similarly, the step's kind and/or name test $v$ yields equality predicates over kind and name (e.g., Lines 6–7).

To provide the relational database management system with complete information about the expected incoming queries, we instructed the compiler to make the semantics of the serialization point ♀ explicit. This adds one extra `descendant-or-self` step to any Query $Q$, originating in its result node sequence:

`for $x in` $Q$ `return $x/descendant-or-self::`*node* ,

where *node* matches all node kinds (including attributes). This produces all XML nodes required to fully serialize the result (surfacing as the additional row variable `d13` in Figure 5.7).

For Query $Q_8$ as an representative of the expected query workload, the DB2 automatic design advisor, `db2advis` [36], suggests the B-tree index set of Table 5.4 (with a total size of 300 MB for a 110 MB instance of the XMark `auction.xml` document). Because of the regularity of the emitted SQL code, the utility of the proposed indexes will be high for any XQUERY workload that exhibits a significant fraction of XQUERY join graphs.

The majority of the index keys proposed in Table 5.4 are prefixed with *low cardinality* column(s), for example, n, nk, or nlk: An XMark XML instance features

| **Index key columns** | **Index deployment** |
|---|---|
| nkspl nlkps nksp nlkp | XPath node test and axis step, access document node (`doc(·)`) |
| vnlkp nlkpv nkdlp | Atomization, value comparison with subsequent/preceding XPath step |
| p\|nvkls | Serialization support (with columns nvkls in DB2's `INCLUDE(·)` clause [36]) |

p:pre, s:pre + size, l:level, k:kind, n:name, v:value, d:data

Table 5.4: B-tree indexes proposed by `db2advis`.

77 distinct element tag and attribute names, regardless of the document size. Similar observations apply to the XML node kinds and the typical XML document height. A B-tree that is organized primarily by such a low cardinality column will, in consequence, *partition* the XML infoset encoding into few disjoint node sets [49]. Note how a name-prefixed index key leads to a B-tree-based implementation of *element tag streams*, the principal data access path used in the so-called *twig join* algorithms [25, 27].

The design advisor further suggests an index with key vnlkp whose value column prefix supports atomization and the general value comparisons between (attribute) nodes featured in Query $Q_8$. A B-tree of this type bears some close resemblance with the XPath-specific indexes (`CREATE INDEX` ... `GENERATE KEY USING XMLPATTERN` ... `AS SQL VARCHAR(`$n$`)`) employed by pureXML™ [14].

### XPath Stitching and Branching

How exactly does DB2's query optimizer deploy the indexes proposed by its design advisor companion? An answer to this question can be found through an analysis of the execution plan generated by the optimizer. We have, in fact, observed a few not immediately obvious "tricks" that have found their way into the execution plans. Most of these observations are closely related to query evaluation techniques that have originally been described as XPath–specific [25,86,94], outside the relational domain.

The optimized DB2 execution plan found for Query $Q_8$ is shown in Figure 5.8.[3] We are reproducing this execution plan in a form closely resembling the output of DB2's Visual Explain facility. Nodes in these plans represent operators of DB2's variant of physical algebra—all operators relevant for the present discussion are introduced in Table 5.5.

We further annotated the execution plan in Figure 5.8 with XPath fragments that indicate which nodes are affected by the corresponding index lookup. The

[3]On XMark sf 1 the query is evaluated within 544 milliseconds, whereas the initial loop lifted variant does not finish within days.

| Operator | Semantics | Operator | Semantics |
|---|---|---|---|
| RETURN | Result row delivery | SORT | Sort rows (+ duplicate row elimination) |
| NLJOIN | Nested-loop join (left leg: outer) | HSJOIN | Hash join (left leg: probe) |
| IXSCAN | B-tree scan | TBSCAN | Temporary table scan |
| nlkps | Index access | DOC | XML infoset table access |

Table 5.5: Relevant IBM DB2 plan operators.

resumption and continuation points ⬚ describe the connection between the various fragments. Consider, for example, the B-tree index with key nkspl: Because of its nk prefix, this index primarily provides support for XPath name and kind tests. Additionally, however, the index delivers the infoset properties spl and thus provides all necessary information to step along *any* XPath axis. The annotation with the XPath fragment

$$\boxed{\phantom{x}}_1\texttt{::closed\_auction}\boxed{\texttt{/child}}_2$$

in Figure 5.8 therefore describes a lookup in the index nkspl that retrieves columns nk to (a) perform the due name and kind test (name = `'closed_auction'` ∧ kind = `ELEM`) and (b) provides enough information to support the resumption point $\boxed{\phantom{x}}_1$ as well as the XPath continuation $\boxed{\texttt{/child}}_2$. DB2 stitches the various matching XPath resumption and continuation points together by means of residual join predicates.

The query optimizer decides on the processing order of the XPath axes based on its "classical" selectivity notion and the availability of eligible access paths: For a B-tree with name-prefixed keys, the relational database system's data distribution statistics capture tag name distribution, whereas value-prefixed keys lead to statistics about the distribution of the (untyped) element and attribute values.

In the case of Query $Q_8$, this enabled the optimizer to decide that the access path nkdlp, directly leading to `price` nodes (key prefix nk) with a typed decimal value of greater than 500 (key column d), is most selective: Only 9,750 of the 4.7 million nodes in the 110 MB XMark XML instance are `price` elements and only a fraction of these have a typed value in the required range.

DB2 builds a hash map on these `price` nodes and probes the map with the `closed_auction` nodes, thus completing the resumption point $\boxed{\phantom{x}}_2$. Further, the hash join operator has its *early-out* flag set (see ⋉ in Figure 5.8) and thus—similar to the original Query $Q_8$—serves only as a predicate filter for the `closed_auction` nodes.

The B-tree index entries provide sufficient context information to allow for arbitrary path processing orders. In the terminology of [86], we have observed

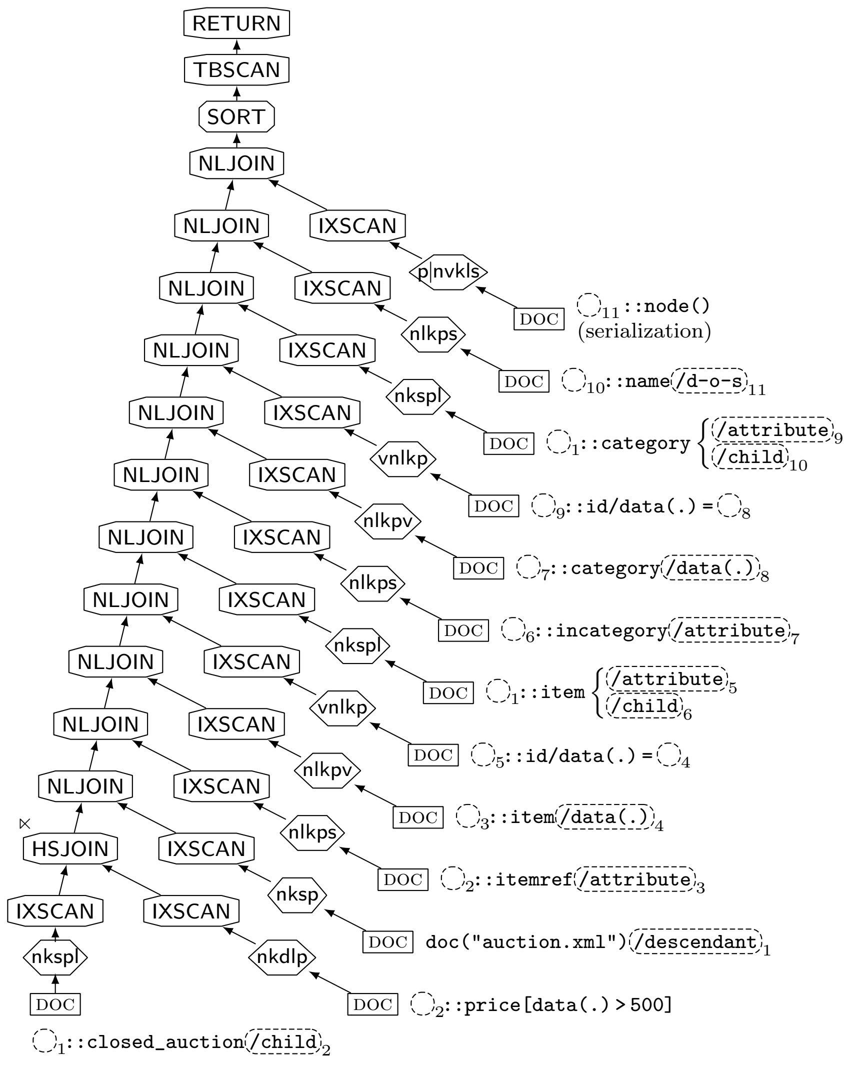


Figure 5.8: DB2 V9 execution plan for Query $Q_8$.

the optimizer to generate the whole variety of *Scan* (strict left-to-right location path evaluation), *Lindex* (right-to-left evaluation), and *Bindex* plans (hybrid evaluation, originating in a context node set established via tag name selection; compare with the initial `closed_auction` node test in Figure 5.8).

The due context $\bigcirc_1$ of the path fragment

$$\bigcirc_1\texttt{::closed\_auction}\boxed{\texttt{/child}}_2\texttt{::price[data(.) > 500]}$$

is provided only by the subsequent NLJOIN–IXSCAN pair, which verifies that the `closed_auction` elements found so far indeed are `descendants` of `auction.xml`'s document node. Observe that, in this specific evaluation order of the location steps, the `closed_auction` nodes now assume the context node role: the plan effectively determines the `closed_auction` elements that have the document node of `auction.xml` in their `ancestor` axis.

In effect, the optimizer mimics a family of rewrites that has been developed in [94]. These rewrites were originally designed to trade reverse XPath axes for their forward duals, which can significantly enlarge the class of expressions tractable by streaming XPath evaluators. Here, instead, we have found the optimizer to exploit the duality in both directions. The evaluation of rooted `/descendant::`$n$ steps—pervasively introduced in [94] to establish a context node set of all elements with tag $n$ in a document—is readily supported by the `n`-prefixed B-tree indexes. Since the XQuery compiler implements the *full axis* feature, it can actually realize a significant fraction of the rewrites in [94].

Finally note how some continuations are used more than once in the downstream execution plan. Continuations with multiple resumption points are the equivalent of the branching nodes discussed in the context of *holistic twig joins* [25, 27, 28]. Quite differently, though, we (a) support the *full axis* feature, (b) let the relational database management system shoulder 100 percent of the evaluation-time and parts of the compile-time effort invested by these algorithms (e.g., the join tree planner implements the *findOrder*(·) procedure of [28] for free), and (c) use built-in B-tree indexes over table-shaped data where *TwigStack* [25] and *Twig*$^2$*Stack* [27] rely on special-purpose runtime data structures, for example, chains or hierarchies of linked stacks and modified B-trees, which call for significant invasive extensions to off-the-shelf database kernels.

### 5.2.3 Loop Lifting an Ordinary Database Workload

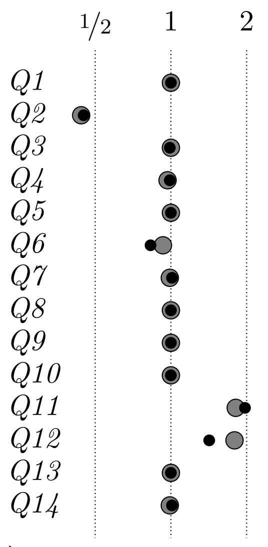
In our previous experiments, we have mainly used XQuery queries to observe the performance impact of the optimizations with respect to the initial loop lifted query plans. Here, we analyze how the combination of loop lifting and optimizations affects an ordinary relational database workload.

We use a SQL-to-SQL compiler that translates SQL queries into algebra plans using a variant of the loop lifted compilation scheme discussed in Chapter 2, optimizes the given plans, and turns these plans back into SQL queries by means of the SQL code generator sketched in Section 3.2. In the context of [58], we use such a SQL-to-SQL compiler to support a declarative language-level debugger. Because of the loop lifted compilation, the debugger can extract *any* meaningful subquery and constructs SQL queries to retrieve the intermediate result computed by that subquery.

To quantify the effect of the loop lifted compilation on query execution time, we executed the original SQL queries *Q1–Q14* of the TPC-H benchmark [121] as well as their loop lifted counterparts against two vanilla TPC-H database instances of scale factor 1 and 10, hosted by DB2. The execution times of the original TPC-H queries, executed as-is as specified by the benchmark, provide the performance baseline for the comparison of Figure 5.9. All timings were measured under the index configuration suggested by DB2's index advisor `db2advis` once it

| **TPC-H** | **sf 1 (1 GB)** | | **sf 10 (10 GB)** | |
|---|---|---|---|---|
| **Query** | original | loop lifted | original | loop lifted |
| *Q1* | 18.20 | 18.23 | 184.29 | 183.49 |
| *Q2* | 0.04 | 0.02 | 0.48 | 0.21 |
| *Q3* | 5.43 | 5.39 | 98.69 | 98.33 |
| *Q4* | 4.43 | 4.40 | 99.29 | 96.30 |
| *Q5* | 1.08 | 1.08 | 23.10 | 23.10 |
| *Q6* | 0.71 | 0.59 | 13.59 | 12.60 |
| *Q7* | 0.48 | 0.49 | 7.25 | 7.17 |
| *Q8* | 0.29 | 0.29 | 5.09 | 5.08 |
| *Q9* | 2.86 | 2.87 | 76.66 | 76.29 |
| *Q10* | 1.07 | 1.07 | 24.86 | 24.95 |
| *Q11* | 0.19 | 0.37 | 2.28 | 4.14 |
| *Q12* | 0.33 | 0.47 | 5.03 | 9.01 |
| *Q13* | 3.25 | 3.27 | 45.41 | 45.55 |
| *Q14* | 0.19 | 0.19 | 2.26 | 2.24 |

**(a)** Query timings (SQL evaluation times, in seconds).



**(b)** Relative execution times (● $\hat{=}$ **sf 1**, ● $\hat{=}$ **sf 10**).

Figure 5.9: A quantification of the performance impact of the loop lifted compilation strategy (measured on DB2).

had been exposed to the original TPC-H workload.

Figure 5.9(b) reports no significant performance impact for most queries.[4] A more thorough comparison of the SQL queries reveals that TPC-H Queries *Q1*, *Q3*, *Q5*, *Q6*, *Q7*, *Q10*, and *Q14* result in almost the same query text: The only difference is that the SQL code generator introduced a common table binding for every aggregate operator. For queries with nesting in the `FROM` clause (Queries *Q8*, *Q9*, and *Q13*), the SQL code generator returned flat queries and only introduced an additional common table expression for the nested aggregate in Query *Q13*.

The loop lifted variant of Query *Q12* replaces an `IN` list used in the original query by a series of `OR` clauses. For the back-end database system DB2, this small modification is enough to choose a different (slower) execution plan.

For the remaining three queries (TPC-H Query *Q2*, *Q4*, and *Q11*), the generated SQL code includes SQL language constructs that were not used in the original query variants. Query *Q2* contains a numbering operator that ensures the alignment of a query block that initially was a nested subquery (`ps_supplycost = (`*subquery*`)`).

TPC-H Query *Q4* contains a correlated subquery (Figure 5.10), which the optimizations were able to resolve: The corresponding loop lifted SQL query (Figure 5.11) features a uncorrelated query that produces a list of duplicate-free order keys (`cte1`). The second table binding `cte2` as well as the top-most query

[4]A mark ○ on the line labeled "2" would indicate that the loop lifted SQL compiler leads to doubled query execution times, for example.

```
select   o_orderpriority,
         count(*) as order_count
from     orders
where    o_orderdate >= date ('1993-07-01')
         and o_orderdate < date ('1993-07-01') + 3 month
         and exists (select   *
                     from     lineitem
                     where    l_orderkey = o_orderkey
                              and l_commitdate < l_receiptdate)
group by o_orderpriority
order by o_orderpriority;
```

Figure 5.10: Original TPC-H Query *Q4*.

```
WITH
-- binding because of duplicate elimination
cte1 (item1) AS
  (SELECT DISTINCT t1.l_orderkey AS item1
     FROM lineitem AS t1
    WHERE t1.l_commitdate < t1.l_receiptdate),

-- binding because of aggregate
cte2 (item2, item3) AS
  (SELECT t2.o_orderpriority AS item2,
          COUNT (*) AS item3
     FROM orders AS t2,
          cte1 AS t3
    WHERE t2.o_orderkey = t3.item1
      AND t2.o_orderdate < date ('1993-07-01') + 3 month
      AND NOT (t2.o_orderdate < date ('1993-07-01'))
    GROUP BY t2.o_orderpriority)

SELECT t4.item2, t4.item3
  FROM cte2 AS t4
 ORDER BY t4.item2 ASC;
```

Figure 5.11: Loop lifted variant of TPC-H Query *Q4*.

that features the `ORDER BY` condition, are again almost identical to the original query.

For TPC-H Query *Q11*, the optimizations could not eliminate all numbering operators thus resulting in a slower query evaluation. Because of a multiplication in the argument of a sum (`sum(ps_supplycost * ps_availqty)`), Rewrites 50 and 51 failed to remove the mapping joins whose columns referenced the numbering operators. As in Section 5.1.1, additional rewrite rules can overcome this limitation.

We conclude that the optimized loop lifted SQL queries are “well-behaved” in many cases and in general do not overwhelm off-the-shelf relational query optimizers.

# Chapter 6

# Summary and Outlook

Pervasive order, data nesting, and the support for tree-structured data were deliberately ignored for over 30 years in most relational database systems. Here, we took the ideas of loop lifting [63] to devise a compilation scheme that transforms side effect free list-based programs with order, nesting, and XML concepts into algebraic queries (Chapter 2). One such supported language is LINQ, which—unlike SQL/XML—provides a natural integration of XML in relational query processing [59].

The queries resulting from the compilation may be evaluated on any SQL:1999 database system or the highly specialized XQUERY back-end MonetDB/XQuery (Chapter 3). Because any database system we had on our workbench struggled with the generated execution plans, this work introduced a logical query optimizer that copes with the unusual plan shape. A rewrite framework that collects operator and column properties performs the logical optimization based on local, peephole-style rewrites (Chapter 4). Three heuristics guide the optimization: They (a) remove superfluous operators, (b) minimize the order constraints, and (c) unnest the query plans.

The outcome are significantly simplified query plans, which often may be automatically turned into a single, flat `SELECT DISTINCT·FROM·WHERE·ORDER BY` SQL query, are robust with respect to semantically equivalent, yet syntactically differing queries, and whose execution times can almost keep up with hand-optimized queries (Chapter 5).

## 6.1 Contributions

This work discusses the compilation, optimization, and evaluation of non-relational queries in the context of relational database systems. The initial compilation (Chapter 2) and the SQL code generation (Section 3.2) are a joint effort with Manuel Mayr, Tom Schreiber, and Jens Teubner. The main focus of this work lies on the optimization of the logical query plans.

### 6.1.1 Optimization of Large Query Plans

The initial query plans are DAG-shaped and feature hundreds of operators. To cope with these large query plans, peephole-style rewrites drive the rewrite framework. The rewrites are supported by a number of properties, which collect various properties of an operator's upstream and downstream plans. Based on this property information only local rewrites are performed.

All properties may be inferred by a single DAG traversal. These properties collect information on keys, constants, abstract domain relationships, functional dependencies, available as well as required columns, the use of columns, expected selection values, duplicates, operator cardinality, and the number of parent operators. While many of these properties have already been described in the context of relational query processing, we adjusted them to cope with the DAG structure and a number of new operators such as the numbering and XML operators.

A common subplan elimination algorithm `cse` additionally supplies a global simplification that merges equivalent subplans irrespective of column naming. For any back-end that supports plan reuse this optimization increases query performance.

### 6.1.2 Order Minimization

Given our unordered logical algebra, the loop lifted compilation encodes order by means of numbering operators. We refined the initial approach that relies on row numbering operators $\vec{\#}$ [63] and introduced the less restrictive rank operators ⛉ to provide relative order information.

We devised a set of rewrite rules that remove superfluous rank operators based on required column information, simplify their arguments with the help of constant, key, and cardinality information, and merge ordering criteria of adjacent ordering operators. To minimize the order constraints, a further set of rewrite rules push up the ⛉ operators until they either merge into a necessary absolute ordering (e.g., a positional predicate) or reach the serialization point to encode the overall output order.

### 6.1.3 Unnesting of Loop Lifted Query Plans

Nested `for` loops as well as the intermediate nesting expressed by `box`/`unbox` pairs lead to query plans with a large number of mapping joins. We devised a set of rewrite rules that take key and domain information into account and push down equi-joins and subsequently eliminate the superfluous ones.

A second set of rewrite rules consumes the resulting long chains of operators, detects plan fragments that are independent of their upstream plan, and unnests these plan fragments. While the former set of rules removes much of the surface query structure, the application of the latter rules results in the detection of value-based joins.

The query unnesting is the key ingredient turning MonetDB/XQuery into an efficient and robust XQuery processor, whose performance—in contrast to its previous version [21]—is not affected by syntactic variations.

### 6.1.4 MonetDB/XQuery and Pathfinder

This document represents an excerpt of the implementation performed in the context of Pathfinder and MonetDB/XQuery. The current version of MonetDB/XQuery constitutes a mature XQuery processor that can process large quantities of XML data in interactive time. In addition to compiling XQuery queries, Pathfinder's code base has been extended to consume and optimize arbitrary logical query plans and to turn the resulting plans into SQL queries. For the non-relational input languages Ferry, Links, Linq, and Ruby this leads to very convincing results.

Improvements to MonetDB/XQuery and Pathfinder that were not discussed in this work include, but are not limited to, (a) XQuery debugging support [60], (b) efficient support for recursive queries [4,5], (c) side-effecting error handling that does not affect the optimizations, (d) fast XML subtree serialization, (e) fast runtime support for node construction,[1] (f) twig constructors replacing multiple node constructors, (g) zero-cost subtree copies for serialize-only subtrees, (h) explicit query result caching, and (i) the integration of DataGuides [47].[2]

## 6.2 Conclusion

We demonstrated that relational database systems are able to efficiently support languages that operate on ordered and nested structures as well as XML data. Our optimizations—that is, the combination of the three heuristics house cleaning, order minimization, and query unnesting—unlock the potential of the relational back-ends. Together with the SQL code generator, we provide a compiler for non-relational query languages that turns a large number of relational database management systems into efficient execution environments for data-intensive programs.

While the optimizations are guided by heuristics only, we almost always observed small improvements. For certain "important" classes of queries such as join graph queries, the optimized variants were orders of magnitudes faster. Choosing an unordered logical algebra furthermore allowed us to generate SQL

[1]MonetDB/XQuery does not use a `printf`-style node construction as in Figure 3.5, but generates new nodes that can be queried within the same query.

[2]DataGuide support is not active in MonetDB/XQuery and was switched off for the generated SQL queries.

queries and thus benefit from orthogonal, cost-based query optimizations already built into off-the-shelf relational query processors.

Finally, the combination of loop lifting and optimizations enabled and facilitated a number of additional research topics such as database-supported program execution in the scope of FERRY, LINKS, LINQ, and RUBY [56, 59, 109, 110], efficient support for recursion [4, 5], declarative language-level debugging [58, 60], runtime query optimization [74, 75], XQUERY full text support [68], XQUERY statistics [119], and, most notably, the high-performance XQUERY processor MonetDB/XQuery.

## 6.3 Open Problems and Future Work

This work started out with a focus on XQUERY processing on large XML documents, before integrating other approaches into the optimization process as well. Here, we review three topics that identify current deficiencies and sketch possible improvements for future Pathfinder versions.

### 6.3.1 More XML Related Optimizations for XQUERY

We described only a few XML-related optimization opportunities. The integration of DataGuide support in MonetDB/XQuery mentioned in Section 4.5 might further speed up MonetDB/XQuery's query processing.

A more interesting optimization, however, would be to merge XPath steps operating on transient nodes with the corresponding node constructors [76]. Such a scenario arises, for example, if a query tries to operate on a tuple-like structure. In XQUERY, tuples can be expressed only by means of XML trees: Node constructors align multiple items and subsequent path steps extract these items again.

To integrate such an optimization, we would need to introduce rewrites that push down ⎍ and ⚛ operators and merge them with the node constructors. Existing properties such as *icols* and *use* and additional properties that record information about nodes (stored vs. transient) and their subtree cardinality (single item per node vs. multiple items per node) can ensure the correctness of the rewrites.

### 6.3.2 Surrogate Values in Optimized Query Plans

Our approach encodes data nesting by means of surrogate values. The numbering operators $\vec{\#}$ and $\vec{⛉}$ provide these values and every invocation of `box` introduces a new surrogate-based foreign-key relationship. The optimizations are able to eliminate the mapping joins and surrogates for combinations of calls to `box` and `unbox`. The optimizations, however, are not able to remove the numbering operators for queries with nested result types, as these surrogates end up in different queries.

In some situations, these remaining numbering operators lead to query plans in which the calculation of cross products is enforced or a database system's index support is effectively disabled. To improve such query plans, Pathfinder would need to optimize multiple query plans simultaneously to consistently resolve the foreign-key relationships.

An alternative approach might be a modification of the compilation scheme. Multiple, already existing columns that form a candidate key could be used as a replacement for the current single surrogate column [122]. This change would result in a significantly smaller number of numbering operators, yet more comparison conditions in the mapping equi-joins. In consequence, however, the equi-join pushdown rewrites would require a major overhaul.

### 6.3.3 Automatization of Rewrites

Although the current version of Pathfinder gives its users a fine-grained control over the optimizations, by default, rewrites are performed in a fixed order and with a predetermined number of repetitions. Both, order and repetitions, were determined incrementally based on the manual observation of hundreds of optimized query plans.

While Pathfinder's optimizations tend to optimize most queries well, we often observed queries where a subset of the optimizations already resulted in the same query plan. But there were also queries whose resulting query plans provided further potential for optimization. Both scenarios indicate that an adaptive query optimization approach might save compilation time for simple queries and improve the query plans for complex queries.

An automated adjustment of the optimizations might furthermore take the query complexity as well as the sizes of the affected documents into account. MonetDB/XQuery, for example, evaluates simple XPath queries on moderately big documents ($< 100\,\text{MB}$) very fast regardless of the optimizations.

# Bibliography

[1] ActiveRecord in Ruby on Rails. `http://ar.rubyonrails.org/`.

[2] Loredana Afanasiev. *Querying XML: Benchmarks and Recursion.* Ph.D. Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, Dec 2009.

[3] Loredana Afanasiev, Massimo Franceschet, Maarten Marx, and Enrico Zimuel. XCheck: A Platform for Benchmarking XQuery Engines. In *Proc. VLDB*, pages 1247–1250, 2006.

[4] Loredana Afanasiev, Torsten Grust, Maarten Marx, Jan Rittinger, and Jens Teubner. An Inflationary Fixed Point Operator in XQuery. In *Proc. ICDE*, pages 1504–1506, 2008.

[5] Loredana Afanasiev, Torsten Grust, Maarten Marx, Jan Rittinger, and Jens Teubner. Recursion in XQuery: Put Your Distributivity Safety Belt On. In *Proc. EDBT*, pages 345–356, 2009.

[6] Loredana Afanasiev, Ioana Manolescu, and Philippe Michiels. MemBeR: A Micro-Benchmark Repository for XQuery. In *Proc. XSym*, pages 144–161, 2005.

[7] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient Optimization of a Class of Relational Expressions. *ACM Transactions on Database Systems (TODS)*, 4(4):435–454, 1979.

[8] Wouter Alink, R. A. F. Bhoedjang, Peter A. Boncz, and Arjen P. de Vries. XIRAF - XML-Based Indexing and Querying for Digital Forensics. *Digital Investigation*, 3(Supplement-1):50–58, 2006.

[9] Wouter Alink, R. A. F. Bhoedjang, Arjen P. de Vries, and Peter A. Boncz. Efficient XQuery Support for Stand-Off Annotation. In *Proc. XIME-P*, 2006.

[10] Wouter Alink, Valentin Jijkoun, David Ahn, Maarten de Rijke, Peter A. Boncz, and Arjen P. de Vries. Representing and Querying Multi-Dimensional Markup for Question Answering. In *Proc. NLPXML*, April 2006.

[11] AMBITION (RUBY). `http://ambition.rubyforge.org/`.

[12] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, M. Holstege, J. Melton, M. Rys, and J. Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text (Working Draft). W3 Consortium, May 2007. `http://www.w3.org/TR/xpath-full-text-10/`.

[13] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *Proc. SIGMOD*, pages 1195–1206, 2008.

[14] Andrey Balmin, Kevin S. Beyer, Fatma Özcan, and Matthias Nicola. On the Path to Efficient XML Queries. In *Proc. VLDB*, pages 1117–1128, 2006.

[15] Catriel Beeri, Ronald Fagin, and John H. Howard. A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations. In *Proc. SIGMOD*, pages 47–61, New York, NY, USA, 1977. ACM.

[16] S. Boag, D. Chamberlin, and M. Fernández. XQuery 1.0: An XML Query Language. W3 Consortium, January 2007. `http://www.w3.org/TR/xquery/`.

[17] P.A. Boncz, M. Zukowski, and N. Nes. X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, pages 225–237, Asimolar, USA, 2005.

[18] Peter A. Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications.* Ph.D. Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[19] Peter A. Boncz, Jan Flokstra, Torsten Grust, Maurice van Keulen, Stefan Manegold, K. Sjoerd Mullender, Jan Rittinger, and Jens Teubner. MonetDB/XQuery-Consistent and Efficient Updates on the Pre/Post Plane. In *Proc. EDBT*, pages 1190–1193, 2006.

[20] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: XQuery—The Relational Way. In *Proc. VLDB*, pages 1322–1325, Trondheim, Norway, August 2005.

[21] Peter A. 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. SIGMOD*, pages 479–490, Chicago, USA, June 2006.

[22] Peter A. Boncz and Martin L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB Journal*, 8(2):101–119, 1999.

[23] Peter A. Boncz, Stefan Manegold, and Jan Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proc. XIME-P*, 2005.

[24] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-Fledged Algebraic XPath Processing in Natix. In *Proc. ICDE*, pages 705–716, 2005.

[25] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD*, pages 310–321, Madison, USA, 2002.

[26] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proc. PODS*, pages 34–43, 1998.

[27] Songting Chen, Hua-Gang Li, Jun'ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig$^2$Stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries over XML Documents. In *Proc. VLDB*, 2006.

[28] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan, and Stelios Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. VLDB*, pages 237–248, 2003.

[29] Sophie Cluet and Guido Moerkotte. Nested Queries in Object Bases. In *Proc. DBPL*, pages 226–242, 1993.

[30] John Cocke. Global Common Subexpression Elimination. *SIGPLAN Notices*, 5(7):20–24, 1970.

[31] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *Proc. FMCO*, pages 266–296, 2006.

[32] George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, pages 268–279, 1985.

[33] John Cowan and Richard Tobin. XML Information Set. W3 Consortium, February 2004. `http://www.w3.org/TR/xml-infoset/`.

[34] H. Darwen. The Role of Functional Dependence in Query Decomposition. In C. J. Date and H. Darwen, editors, *Relational Database: Writings 1989-1991*, pages 133–154. Addison-Wesley, Reading, MA, 1992.

[35] Jack W. Davidson and Christopher W. Fraser. The Design and Application of a Retargetable Peephole Optimizer. *ACM TOPLAS Journal*, 2(2):191–202, April 1980.

[36] DB2 9 for Linux, UNIX and Windows Manuals, 2007. `http://www.ibm.com/software/data/db2/udb/`.

[37] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, January 2007. `http://www.w3.org/TR/xquery-semantics/`.

[38] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 4th edition, July 2003.

[39] ADO.NET Entity Framework. `http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx`.

[40] Ronald Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems (TODS)*, 2(3):262–278, 1977.

[41] Mary F. Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions. In *Proc. DEXA*, pages 554–563, Copenhagen, Denmark, 2005.

[42] The Ferry Query Compiler. `http://www.ferry-lang.org/`.

[43] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a Native XML Base Management System. *VLDB Journal*, 11(4):292–314, 2002.

[44] Patrick C. Fischer and Stephen J. Thomas. Operators for Non-First-Normal-Form Relations. In *Proc. COMPSAC*, pages 464–475, 1983.

[45] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, September 1999.

[46] César A. Galindo-Legaria and Milind Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *Proc. SIGMOD*, pages 571–581, 2001.

[47] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization. In *Proc. VLDB*, pages 436–445, Athens, Greece, August 1997.

[48] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[49] Goetz Graefe. Sorting and Indexing with Partitioned B-Trees. In *Proc. CIDR*, 2003.

[50] Torsten Grust. Accelerating XPath Location Steps. In *Proc. SIGMOD*, pages 109–120, 2002.

[51] Torsten Grust. Purely Relational FLWORs. In *Proc. XIME-P*, Baltimore, MD, USA, June 2005.

[52] Torsten Grust and Stefan Klinger. Schema Validation and Type Annotation for Encoded Trees. In *Proc. XIME-P*, pages 55–60, 2004.

[53] Torsten Grust, Manuel Mayr, and Jan Rittinger. XQuery Join Graph Isolation. In *Proc. ICDE*, pages 1167–1170, 2009.

[54] Torsten Grust, Manuel Mayr, and Jan Rittinger. Let SQL Drive the XQuery Workhorse. In *Proc. EDBT*, 2010.

[55] Torsten Grust, Manuel Mayr, Jan Rittinger, Sherif Sakr, and Jens Teubner. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proc. SIGMOD*, pages 1162–1164, 2007.

[56] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-Supported Program Execution. In *Proc. SIGMOD*, pages 1063–1066, 2009.

[57] Torsten Grust and Jan Rittinger. Jump Through Hoops to Grok the Loops—Pathfinder's Purely Relational Account of XQuery-style Iteration Semantics. In *Proc. XIME-P*, Vancouver, Canada, June 2008.

[58] Torsten Grust and Jan Rittinger. Observing SQL Queries in their Natural Habitat, 2010. Submitted.

[59] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-Safe LINQ Compilation. In *Proc. VLDB*, September 2010.

[60] Torsten Grust, Jan Rittinger, and Jens Teubner. Data-Intensive XQuery Debugging with Instant Replay. In *Proc. XIME-P*, Beijing, China, June 2007.

[61] Torsten Grust, Jan Rittinger, and Jens Teubner. eXrQuy: Order Indifference in XQuery. In *Proc. ICDE*, pages 226–235, Istantbul, Turkey, April 2007. IEEE Computer Society.

[62] Torsten Grust, Jan Rittinger, and Jens Teubner. Why Off-the-Shelf RDBMSs are Better at XPath Than You Might Expect. In *Proc. SIGMOD*, pages 949–958, 2007.

[63] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, pages 252–263, 2004.

[64] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Proc. Twente Data Management Workshop*, pages 9–16, Enschede, The Netherlands, June 2004.

[65] Torsten Grust and Maurice van Keulen. Tree Awareness for Relational DBMS Kernels: Staircase Join. In *Proc. Intelligent Search on XML Data*, pages 231–245, 2003.

[66] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. VLDB*, pages 524–535, Berlin, Germany, September 2003.

[67] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems (TODS)*, 29:91–131, 2004.

[68] Djoerd Hiemstra, Henning Rode, Roel van Os, and Jan Flokstra. PF/Tijah: Text Search in an XML Database System. In *Proc. OSIR*, August 2006.

[69] Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML). I.O. for Standardization (ISO).

[70] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. An Architecture for Recycling Intermediates in a Column-Store. In *Proc. SIGMOD*, pages 309–320, 2009.

[71] 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. *VLDB Journal*, 11(4):274–291, 2002.

[72] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[73] Simon Peyton Jones and Philip Wadler. Comprehensive Comprehensions: Comprehensions with "Order by" and "Group by". In *Proc. ACM SIGPLAN workshop on Haskell*, pages 61–72, 2007.

[74] Riham Abdel Kader, Peter A. Boncz, Stefan Manegold, and Maurice van Keulen. ROX: Run-Time Optimization of XQueries. In *Proc. SIGMOD*, pages 615–626, 2009.

[75] Riham Abdel Kader, Peter A. Boncz, Stefan Manegold, and Maurice van Keulen. ROX: The Robustness of a Run-Time XQuery Optimizer Against Correlated Data. In *Proc. ICDE*, pages 1185–1188, 2010.

[76] Hiroyuki Kato, Soichiro Hidaka, Zhenjiang Hu, Keisuke Nakano, and Yasunori Ishihara. Context-Preserving XQuery Fusion. In *Proc. APLAS*, November 2010.

[77] Martin L. Kersten and Stefan Manegold. Cracking the Database Store. In *Proc. CIDR*, pages 213–224, Asimolar, USA, 2005.

[78] Setrag Khoshafian, George P. Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A Query Processing Strategy for the Decomposed Storage Model. In *Proc. ICDE*, pages 636–643, 1987.

[79] Anthony C. Klug. Calculating Constraints on Relational Expressions. *ACM Transactions on Database Systems (TODS)*, 5(3):260–290, September 1980.

[80] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. VLDB*, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[81] LINQ Standard Query Operators. `http://msdn.microsoft.com/en-us/library/bb397896.aspx`.

[82] LINQ to SQL: .NET Language-Integrated Query for Relational Data. `http://msdn.microsoft.com/en-us/library/bb425822.aspx`.

[83] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3 Consortium, January 2007. `http://www.w3.org/TR/xquery-semantics/`.

[84] Sabine Mayer, Torsten Grust, Maurice van Keulen, and Jens Teubner. An Injection of Tree Awareness: Adding Staircase Join to PostgreSQL. In *Proc. VLDB*, pages 1305–1308, 2004.

[85] Manuel Mayr. *A SQL:99 Code Generator for Pathfinder*. Master Thesis, Technische Universität München, Munich, Germany, Apr 2007.

[86] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. VLDB*, pages 315–326. Morgan Kaufmann, Sep 1999.

[87] William M. McKeeman. Peephole Optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[88] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Objects, Relations, and XML in the .NET Framework. In *Proc. SIGMOD*, pages 706–706, 2006.

[89] Jim Melton. *Advanced SQL:1999: Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann Publishers, Amsterdam, 2003.

[90] Thomas Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs.* Dissertation, Universität Mannheim, July 2005.

[91] Thomas Neumann and Guido Moerkotte. A Combined Framework for Grouping and Order Optimization. In *Proc. VLDB*, pages 960–971, Toronto, Canada, September 2004. Morgan Kaufmann Publishers.

[92] Thomas Neumann and Guido Moerkotte. Generating Optimal DAG-Structured Query Evaluation Plans. *Computer Science - R&D*, 24(3):103–117, 2009.

[93] Atsushi Ohori. Type-Directed Specialization of Polymorphism. *Information and Computation*, 155(1-2):64–107, 1999.

[94] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proc. XML-Based Data Management and Multimedia Engineering*, pages 109–127, Prague, Czech Republic, March 2002.

[95] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. SIGMOD*, pages 903–908, 2004.

[96] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Peter Kukol, Wei Yu, Dragan Tomic, Adrian Baras, Chris Kowalczyk, Brandon Berg, Denis Churin, and Eugene Kogan. XQuery Implementation in a Relational Database System. In *Proc. VLDB*, pages 1175–1186, Trondheim, Norway, August 2005.

[97] Stelios Paparizos and H. V. Jagadish. Pattern Tree Algebras: Sets or Sequences? In *Proc. VLDB*, pages 349–360, Trondheim, Norway, 2005.

[98] Simon Peyton Jones. The Haskell 98 Language. *Journal of Functional Programming*, 13(1), 2003.

[99] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD*, pages 39–48, New York, NY, USA, 1992. ACM.

[100] The Python Programming Language. `http://www.python.org/`.

[101] Ruby on Rails. `http://rubyonrails.org/`.

[102] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[103] Jan Rittinger, Jens Teubner, and Torsten Grust. Pathfinder: A Relational Query Optimizer Explores XQuery Terrain. In *Proc. BTW*, pages 617–620, 2007.

[104] Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. XQuery 1.1: An XML Query Language (Working Draft). W3 Consortium, December 2009. `http://www.w3.org/TR/2009/WD-xquery-11-20091215/`.

[105] The Ruby Programming Language. `http://www.ruby-lang.org/`.

[106] Hans-Jörg Schek and Marc H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2):137–147, 1986.

[107] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, pages 974–985, 2002.

[108] Joachim W. Schmidt. Some High-Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems (TODS)*, 2(3):247–261, 1977.

[109] Tom Schreiber. *Translation of List Comprehensions for Relational Database Systems.* Master Thesis, Technische Universität München, Munich, Germany, Mar 2008.

[110] Tom Schreiber, Simone Bonetti, Torsten Grust, Manuel Mayr, and Jan Rittinger. Thirteen New Players in the Team: A Ferry-Based LINQ to SQL Provider. In *Proc. VLDB*, September 2010.

[111] 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. SIGMOD*, pages 23–34, Boston, USA, May 1979.

[112] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts.* McGraw-Hill, New York, San Francisco, Washington, DC, USA, 5th edition, May 2005.

[113] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental Techniques for Order Optimization. In *Proc. SIGMOD*, pages 57–67, 1996.

[114] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Bringing Order to Query Optimization. *SIGMOD Record*, 31(2):5–14, 2002.

[115] James F. Terwilliger, Philip A. Bernstein, and Sergey Melnik. Full-Fidelity Flexible Object-Oriented XML Access. In *Proc. VLDB*, volume 2, pages 1030–1041, 2009.

[116] James F. Terwilliger, Sergey Melnik, and Philip A. Bernstein. Language-Integrated Querying of XML Data in SQL Server. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1396–1399, 2008.

[117] Jens Teubner. Pathfinder: XQuery Compilation for Relational Database Targets. In *Proc. BTW*, pages 465–474, 2007.

[118] Jens Teubner. Scalable XQuery Type Matching. In *Proc. EDBT*, pages 38–48, Nantes, France, March 2008.

[119] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable Cardinality Forecasts for XQuery. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):463–477, 2008.

[120] Jens T. Teubner. *Pathfinder: XQuery Compilation for Relational Database Targets.* Dissertation, Technische Universität München, September 2006.

[121] TPC Benchmark H. T. P. P. Council. `http://www.tpc.org/tpch/`.

[122] Jan van den Bussche. Simulation of the Nested Relational Algebra by the Flat Relational Algebra, with an Application to the Complexity of Evaluating Powerset Algebra Expressions. *Theoretical Computer Science*, 254(1-2):363–377, 2001.

[123] Philip Wadler. Comprehending Monads. In *Proc. ACM conference on LISP and functional programming*, pages 61–78, 1990.

[124] Song Wang, Elke A. Rundensteiner, and Murali Mani. Optimization of Nested XQuery Expressions with Orderby Clauses. *Data & Knowledge Engineering*, 60(2):303–325, Feb 2007.

[125] Xiaoyu Wang and Mitch Cherniack. Avoiding Sorting and Grouping in Processing Queries. In *Proc. VLDB*, pages 826–837, 2003.

[126] Ying Zhang and Peter A. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *Proc. VLDB*, pages 99–110, 2007.

[127] Ying Zhang and Peter A. Boncz. XRPC: Distributed XQuery and Update Processing with Heterogeneous XQuery Engines. In *Proc. SIGMOD*, pages 1331–1336, 2008.

[128] Ying Zhang, Nan Tang, and Peter A. Boncz. Efficient Distribution of Full-Fledged XQuery. In *Proc. ICDE*, pages 565–576, 2009.

# Acknowledgments

The outcome of my master project was the first substantial back-end code generator for Pathfinder, which compiles XQuery queries directly into MonetDB's MIL code. While this code generator included a syntactic join detection logic and led to the first successful version of MonetDB/XQuery [21], I was disappointed that already slight tweaks to the query text led to huge performance penalties. I thus longed for a better solution. Torsten Grust gave me the opportunity to unlock MonetDB/XQuery's potential and offered me a job in his team at the TU München, which I gladly accepted. I still do not regret that decision!

During my time at the TU München and the Universität Tübingen, Torsten's office was always open, which I shamelessly exploited not only to get feedback, but also to show off the newest improvement. He supported me in most of my decisions and guided me, whenever I seemed to reach a dead end. Torsten, thank you very much for everything.

Further, I would like to thank Jens Teubner. With his intuition to explain even the most complex problem in an intelligible way, he could very often help me—especially at the beginning of my work. His thorough attitude was stimulating in many situations and strongly influenced me.

The improvement of MonetDB/XQuery was my driving force in the first years. The interest and support of Peter Boncz and Stefan Manegold as well as Sjoerd Mullender, Lefteris Sidirourgos, Yhing Zhang and Henning Rode from the CWI Amsterdam and Jan Flokstra, Maurice van Keulen, and Riham Abdel Kader from the University of Twente motivated me all the time. Loredana Afanasiev, furthermore, caused me to look at Pathfinder's compilation from a different angle during our joint effort to efficiently evaluate recursive queries in XQuery efficiently. Thanks to all of you.

In the later years, the focus of Pathfinder shifted to support other non-relational query languages. I would like to thank Manuel Mayr and Tom Schreiber for many discussions that led to ever more language concepts the loop lifted compilation was able to support.

During all the time my family gave me constant support: They supported my decision to spend more time at the university, endured two moves into new cities, never failed to believe that I finish this work, and ensured that I kept the balance between work and relaxation. Thanks a million and I love all of you!