An Algebraic Approach to XQuery Optimization

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Wirtschaftsinformatiker
Norman May
aus Leipzig

Mannheim, 2007

Dekan: Professor Dr. Matthias Krause, Universität Mannheim
Referent: Professor Dr. Guido Moerkotte, Universität Mannheim
Korreferent: Professor Dr. Torsten Grust, Technische Universität München

Tag der mündlichen Prüfung: 29. November 2007

Zusammenfassung

Mit der zunehmenden Verbreitung von XML-Daten und XML-Anwendungen wird es wichtiger, XML-Anfragen effizient auszuwerten. Während in den letzten dreißig Jahren eine Reihe von Optimierungstechniken für relationale Datenbanken entwickelt wurden, müssen bei der Optimierung von XML-Anfragen neue Herausforderungen gelöst werden. Insbesondere müssen Optimierer für XQuery, die Standardanfragesprache für XML, sowohl die Dokumentreihenfolge als auch die Sequenzreihenfolge beachten. Andererseits haben sich algebraische Optimierungen in relationalen Datenbanken als flexibel und leistungsfähig erwiesen.

Daher wird in dieser Dissertation ein algebraischer Ansatz für die Optimierung von XQuery-Anfragen entwickelt, der eine einfache Übersetzung von XQuery in diese Algebra ermöglicht. Basierend auf der formalen Definition der algebraischen Operatoren werden Eigenschaften der Algebra formal bewiesen. In dieser Arbeit nutzen wir die Algebra, um algebraische Äquivalenzen für das Entschachteln geschachtelter XQuery-Anfragen zu entwickeln. Nach der Entschachtlung der Anfragen werden nahezu alle Anfragen in Sekunden oder Millisekunden ausgewertet, während die ursprüngliche geschachtelte Anfrage oft mehrere Stunden für die Auswertung benötigt. In dieser Dissertation werden drei Grundmuster für algebraische Äquivalenzen identifiziert. Für die Auswahl der effektivsten Entschachtlungsäquivalenz wird für jedes dieser Grundmuster ein Entscheidungsbaum entwickelt.

Ein weiteres wichtiges Ergebnis der Anfrageentschachtlung besteht darin, dass in der darauf folgenden kostenbasierten Optimierung mehr alternative Pläne, und vor allem meist auch schneller auswertbare Pläne, generiert werden können. In dieser Arbeit werden zwei weitere Fälle präsentiert, in denen der Suchraum für alternative Pläne erweitert werden muß, um effiziente Auswertungspläne zu generieren: das Umordnen von Joins und von Location Steps in Pfadausdrücken. Das in dieser Arbeit vorgestellte algebraische Rahmenwerk erkennt alle Fälle, in denen bei der Umordnung dieser Operationen die Ordnungssemantik von XQuery verletzt wird. Allerdings ermöglichen es aktuelle Ansätze zur Optimierung der Reihenfolge in Anfragen, effizient die korrekte Reihenfolge wieder herzustellen.

Der in dieser Dissertation vorgestellte Ansatz zur algebraischen Optimierung von XQuery stellt somit einen wesentlichen Baustein für die effiziente Auswertung von XML-Anfragen dar. Darüberhinaus profitiert auch Anfrageauswertung in relationalen Datenbanken von diesen Techniken, wenn die Reihenfolge bei der Optimierung berücksichtigt werden muss.

Abstract

As more data is stored in XML and more applications need to process this data, XML query optimization becomes performance critical. While optimization techniques for relational databases have been developed over the last thirty years, the optimization of XML queries poses new challenges. Query optimizers for XQuery, the standard query language for XML data, need to consider both document order and sequence order. Nevertheless, algebraic optimization proved powerful in query optimizers in relational and object oriented databases. Thus, this dissertation presents an algebraic approach to XQuery optimization.

In this thesis, an algebra over sequences is presented that allows for a simple translation of XQuery into this algebra. The formal definitions of the operators in this algebra allow us to reason formally about algebraic optimizations. This thesis leverages the power of this formalism when unnesting nested XQuery expressions. In almost all cases unnesting nested queries in XQuery reduces query execution times from hours to seconds or milliseconds. Moreover, this dissertation presents three basic algebraic patterns of nested queries. For every basic pattern a decision tree is developed to select the most effective unnesting equivalence for a given query.

Query unnesting extends the search space that can be considered during cost-based optimization of XQuery. As a result, substantially more efficient query execution plans may be detected. This thesis presents two more important cases where the number of plan alternatives leads to substantially shorter query execution times: join ordering and reordering location steps in path expressions. Our algebraic framework detects cases where document order or sequence order is destroyed. However, state-of-the-art techniques for order optimization in cost-based query optimizers have efficient mechanisms to repair order in these cases.

The results obtained for query unnesting and cost-based optimization of XQuery underline the need for an algebraic approach to XQuery optimization for efficient XML query processing. Moreover, they are applicable to optimization in relational databases where order semantics are considered.

Acknowledgments

I would like to thank my supervisor, Guido Moerkotte, for encouraging me to engage in this research on optimization and efficient execution of XQuery. His steady support and advice both as a researcher and teacher was invaluable for carrying out this work.

Carl-Christian Kanne served as my mentor when I started working in the Natix project. His guidance in the early phases of this thesis helped me to get acquainted with Natix and to identify challenges to attack. As one of the founders of this project, I am grateful for all the effort he spent on Natix. Being a member of the second generation of PhD students working on Natix, I could benefit from the foundations built by Guido Moerkotte, Carl-Christian Kanne, Till Westmann, Thorsten Fiebig, and Sven Helmer. Sven Helmer was very supportive as a coauthor of several of the papers that emerged from our investigations and as a colleague. It was a great pleasure to work with Alexander Böhm, Matthias Brantner, Thomas Neumann, and Robert Schiele. Discussing ideas with them has always been very instructive. I also enjoyed working with the students that directly or indirectly contributed to this work: Denis Lutz, Robin Aly, Georg Hackenberg, and Susanne Bitzer. I am grateful to Simone Seeger for her careful proof-reading both of the thousands of lines of text I have written in the recent years.

I appreciate Jérôme Siméon and Mary Fernández for starting Galax. When I was in doubt about the XQuery semantics, Galax would always help. Mary encouraged us to extend the ICDE paper on query unnesting into an article for ACM TODS. After diving even deeper into this subject, we found out that we had discovered only the tip of the iceberg yet. I would also like to thank Torsten Grust for serving as the second examiner of this thesis. Torsten together with the MonetDB/XQuery team always turned out to be open-minded and warm-hearted colleagues in this competitive research community.

Finally and most importantly, I owe a debt of gratitude to my wife Jennifer and my family for their constant support. Especially, Jennifer's emotional support and encouragement was essential when deadlines were approaching or when I was in doubt. On the other hand, I am glad to be able to share all the joy with her during this project and in the time thereafter.

Contents

1. Introduction

1.1. Motivation

The Extensible Markup Language (XML) has emerged as one backbone for information processing on the Web, and in business and scientific applications. Ever increasing amounts of XML data are produced, exchanged, stored and analyzed. Many tools have been and still are developed to support these tasks. Most importantly, analyzing and transforming these huge amounts of XML data necessitated the standardization of query languages and transformation languages for XML data. One prominent example is XQuery. Since XQuery queries are executed on ever larger collections of XML documents, query processing must be carried out efficiently. With the research on XQuery processing reported in this thesis, we contribute important building blocks to meet this challenge.

1.1.1. A Brief History of XQuery

Initial research on querying semistructured data[1] laid the foundation for early drafts of XQuery. After almost a decade of work, the W3C published the recommendation of XQuery version 1.0. The specification process sparked new interest in developing (XQuery) query processors. A number of almost complete implementations of the standard were available in sync with its release. Interesting research prototypes were in development, and commercial database vendors extended their relational databases to support XQuery.[2]

Research on XQuery concentrated on efficient storage of XML and evaluation and optimization techniques for XQuery. An increasing number of applications rely on XML data and demand a complete coverage of the standardized features in XQuery. To motivate the need for our research on optimization and efficient execution of XQuery, we investigate the processing requirements of typical XQuery applications.

1.1.2. XQuery Applications

XML Warehouses and XOLAP XML has been adopted for logging events. Usually, the structure of log entries evolves over time, and log entries may describe complex log events. Such loosely structured data is the prime target of XML and XQuery. The structure of log entries is self-describing because tags used in the log entries encode schema information. Complex log events can be represented in XML by nested elements. Because log files easily grow into gigabytes of size, they must be managed by XML database systems. Notice that relational databases do not support this scenario well because they assume unstructured data (i.e. tuples) and a schema that rarely changes. When analyzing the logs, it is important to employ efficient retrieval and transformation algorithms. As queries in XQuery can express complex structural patterns, join conditions, grouping, and aggregation, XQuery is used to analyze such logs.

Once the queries involved become non-trivial, it is not sufficient to simply interpret these XQueries or map them to a standard-evaluation strategy. Instead, an XQuery processor should apply several optimization steps. In the scenario outlined above, order information

[1] Important predecessors of XQuery are Lorel [AQM+97], UnQL [BFS00], the TSIMMIS project [PGMW95], XML-QL [DFF+98], Quilt [CRF00], and XPath 1.0 [CD99].

[2] For example [RSF06, BGvK+06, Kay07, JAKC+02, FHK+02, NDM+01, Sch01, FHK+04, LKA05, PCS+05, OCP+05, NdL05, Tec07].

is relevant: When the temporal order of log entries is encoded in the textual order in the XML document, a query must respect order. On the other hand, aggregate functions are usually insensitive to order and, thus, may enable many optimizations. Query unnesting and join ordering are other key optimizations that an XQuery optimizer must consider in analytical queries.

Several experiments presented in this thesis support the need for powerful optimizers for XQuery. But the techniques developed in this thesis improve the state-of-the-art processing techniques for XQuery by orders of magnitude. For example, we introduce a powerful unnesting framework that often improves query execution times by a factor of 100. We investigate the problem of reordering joins and XPath location steps. This includes formal proofs when reordering operators is valid. But we also discuss how our cost-based query optimizer enumerates valid operator orders and finds the best plan alternative among all these alternatives. We argue that our optimizations should become the core optimization techniques of every XQuery processor.

Information Integration Information integration was one of the first applications and main motivations for the development of XQuery. In information integration, heterogenous data sources are represented as XML views, and their real presentation format is hidden. Queries in applications developed in this context are formulated in XQuery and access these XML views. At the local data sources, XQuery is either implemented using adaptors, cursors, or in the native query language of the data sources, e.g. SQL.

The XQuery processor in such an application needs to decide which data source contributes to the result of a query. The order of the accesses on data sources usually depends on the size of the accessed data and the processing speed of the data source. Moreover, some computations can be pushed to the local databases where expensive processing tasks can often be evaluated more efficiently. Several such non-trivial optimizations assure an efficient processing strategy to evaluate the XQuery query over the heterogenous data sources. The techniques developed in this thesis contribute to this difficult endeavor because we treat XQuery optimization on the algebraic level. For example, we propose algebraic rewrites that merge query blocks into larger ones. Since the cost-based query optimizer works on the level of query blocks, merging them gives the optimizer complete information about the query. Thus, the optimizer can exploit more information when it chooses one of the processing strategies outlined above.

Distributed Processing The core of distributed business processes consists of data and remote procedure calls (RPC). When business processes cross the boundaries of enterprises, it has become customary to encode both the data and the RPCs in XML messages. In a typical communication between partners in a business process, one has to perform the following tasks: (1) analyze the exchanged messages using XPath or XQuery, (2) transform and internally process the data and message, and (3) finally generate replies encoded in XML. Since all these steps are well-supported by XQuery, efforts are under way to implement them by extending XQuery with new processing primitives. The advantages of this integrated support for storing, querying, and transforming XML messages include: (1) Less transcoding between business objects and XML is needed. (2) Optimization opportunities arise when different processes interact or share computations. (3) Access to persistent data can be combined with the the involved transformations. Evidently, the more complex the application scenarios get, the more important it becomes to optimize the embedded XQuery statements. Since lots of business data is managed by relational databases, this data is increasingly complemented by XML data. Hence, integrated processing of XML and relational data is mandatory. Consequently, XQuery processing architectures that fit into the architecture of relational databases are desirable because first, it is possible to leverage techniques developed for relational databases, and second, it is easy to integrate processing of XML and relational data. Our algebraic approach to XQuery optimization

and processing is motivated by these two observations. Our XQuery query optimizer can reuse the architecture from relational query optimizers. Moreover, we were able to keep large parts of our query optimizer independent of the query language. This allows us to integrate optimization of SQL and XQuery into a single query optimizer. In this thesis, we point out where we had to extend our query optimizer specifically for XML query processing and where we could reuse the common architecture of relational query optimizers.

The Web 2.0 As more and more XML dialects and XML applications emerge, e.g. SVG, SMIL, KML in Google Earth, or even as a storage format for office documents, a new trend of integrating them into *Mashups* has emerged. In this area, XQuery is a candidate for replacing script code that is hard to maintain by declarative and optimizable XQuery code. Thus, XQuery may complement access to relational data via dynamic SQL in these applications with access to and integration of diverse XML sources. Users automatically benefit from new optimizations developed for XQuery without sacrificing code maintainability. Thus, XQuery is perceived both as a query language and a scripting language: it gets integrated into programming languages, and it allows extensions via function library modules, similar to user-defined functions in SQL. Extending XQuery into a programming or scripting language is already under discussion [CCF+06].

Publishing 2.0 Currently, publishers are shifting to XML to store their raw content, replacing XML's ancestor SGML [Hun06]. So far, XSLT was used to transform the content stored in an XML document into a publication specifically tailored for the target audience. Now, XSLT is replaced by XQuery because it offers the chance to remove layers from the multi-tier publishing architectures. In this streamlined architecture, XQuery processors are the core component to implement the content logic. Thus, these content management systems use XQuery to select, transform, and combine content. Due to its integral role in this architecture, efficient XQuery processing becomes a key issue. Naturally, publishers have a strong focus in textual content. XQuery provides powerful functions to process text data. But most importantly, it respects document order which is a key requirement for such data. In this thesis, we argue that preserving order at all stages of query processing can be quite costly. Instead, we propose to destroy document order temporarily to repair it later. In our cost-based algebraic framework, this decision leads to query processing strategies superior to current XQuery evaluators.

1.1.3. Observations

As XML applications become more demanding, queries over XML data become more complex and expensive to evaluate. XQuery will only succeed in penetrating application areas as the ones outlined above if it is evaluated fast. We expect that complex constructs, such as node constructors, nested queries, or data retrieval on (several) huge document instances are the prime targets for optimization techniques. In this thesis, we undertake a significant effort to develop an architecture for XQuery optimization and several concrete optimization techniques.

1.2. Natix

Our implementations are integrated into and extend Natix, a native XML database management system [Kan02, FHK+02] developed by our group. Thus, we briefly survey the components of Natix which are relevant for this thesis.

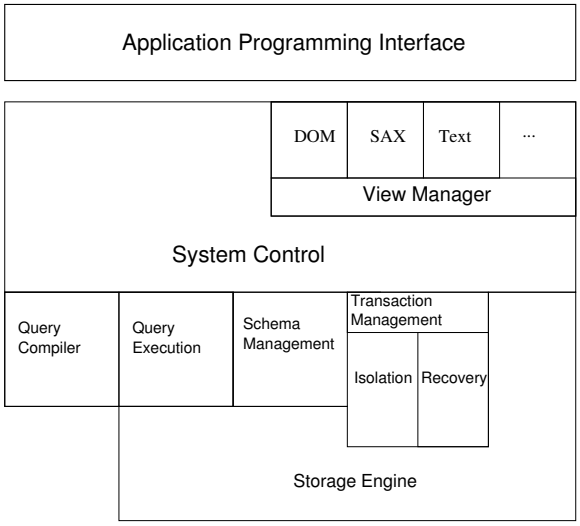


Figure 1.1.: The Natix Architecture

1.2.1. General Architecture

In this thesis, we focus on the optimization and execution of XQuery. The query compiler and the query execution engine are both core components of the Natix system shown in Figure 1.1.

Natix consists of three layers. The bottom layer contains the storage engine including buffer management, see [KM00] for details. The middle layer contains the services one typically expects from a database management system (DBMS). Among these services are the query compiler (QC) and the query execution engine (QEE), which are the main subjects of this thesis. The top layer focuses on system control and provides the interface to the system via a C++ library [BBK+06]. Applications, like the interactive shell included in the Natix distribution, are developed by using this interface.

When formulating queries, we have two alternatives. First, an XQuery query can be passed as a string parameter via the C++ API to the Natix core. This is similar to dynamic SQL. Second, ad-hoc queries can be evaluated within the interactive shell. In both cases, the query is passed to the QC, where a query evaluation plan (QEP) is generated. The QEE then evaluates the QEP and returns the result.

1.2.2. The Query Execution Engine

The query execution engine consists of an iterator-based implementation of algebraic operators. They process ordered sequences of tuples. Tuple attributes either hold base type values such as strings, numbers, and tree node references, or again contain ordered sequences. The iterator model has been slightly extended to deal more efficiently with group boundaries and nested queries.

Subscripts of the algebraic operators (such as join or selection predicates) are expressed in an assembly-like language, and are evaluated using the Natix Virtual Machine (NVM). The NVM avoids the overhead associated with interpreted operator trees.

XML data is accessed through special NVM commands which directly access a clustered persistent XML storage format in the page buffer. Hence, expensive representation changes such as pointer swizzling of the data during query execution are not required. Moreover, our compact format results in few page and CPU cache misses, and performs better than pure pointer-based main memory representations.

1.2.3. The Query Compiler

The architecture of the query compiler is shown in Figure 1.2. It follows the rather traditional six-phase approach. The chapters of this thesis follow the structure of the query compiler.

After the parser has generated an abstract syntax tree in the first step, the NFST module performs **N**ormalization, **F**actorization of common subexpressions, **S**emantic analysis, and **T**ranslation into an internal representation. This internal representation is a mixture of our algebra and a calculus representation. We discuss this module in Chapter 3.

After that, we can start rewriting the queries. Most importantly, we inline views (which are called functions in XQuery), unnest queries, and rewrite XPath expressions. Expanding views can be thought of as replacing a non-recursive function call with the body of the function. Since a nested query results in a nested algebraic expression which in turn requires an inefficient nested-loop evaluation, we try to unnest queries whenever possible. Query unnesting is the main optimization we present in Chapter 4.

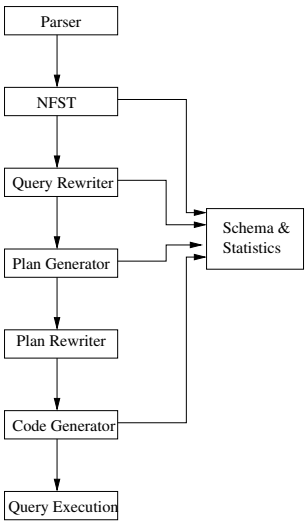


Figure 1.2.: The Natix Query Compiler

During the plan generation phase, which is the subject of Chapter 5, we replace the calculus representation of query blocks with algebraic expressions. Here, the plan generator is faced with numerous alternative QEPs because many execution orders as well as an actual implementation of the algebraic operators are valid but have widely different costs. The plan generator picks the cheapest among all valid query execution plans. Dynamic programming is the prevalent approach when generating execution plans.

In the last but one phase, the generated plan is rewritten. Typically, only small changes are made to the plan. For example, two successive selections are merged. Finally, we generate the code for the QEP. We will treat these two phases briefly in Section 5.5

1.3. Research Objectives

Our observations on evolving XQuery applications clearly demonstrate that efficient XQuery processing is a key requirement. But a direct implementation and execution of queries as defined in the XQuery specification leads to unsatisfactory performance and poor resource utilization. Instead, we aim to develop optimizations for XQuery which satisfy the following four demands: correctness, efficiency, effectiveness, and extensibility. These goals will be used to benchmark our techniques, and we will reiterate them in every chapter.

Correctness Clearly, optimizations need to preserve the semantics of a query. We express most optimizations as equivalences on algebraic expressions. The equivalence transforms a query into a different but, presumably, more efficient query. Evidently, we have to prove the correctness of every equivalence.

Efficiency Users observe the impact of optimizations as improved performance. The performance can be measured in different metrics. Typical metrics to assess the efficiency of query processing include: (1) the time to return the complete query result to the user or application, (2) the time to compute the first k results, (3) low resource consumption during optimization and query processing, (4) high overall resource utilization of the system that processes the query. Since several of these objectives conflict, we have to prioritize them.

In this thesis, we focus on (1), i.e. we want to minimize the time to compute the complete query result. In most cases, this also leads to good resource utilization.

Effectiveness Clearly, query optimization comes at some cost. Thus, we require that the cost for optimization must pay off during query execution. To show this, we conduct experiments that assess query optimization and execution times.

Extensibility As more optimizations for XQuery are developed, the query optimizer needs to incorporate them into an overall efficient XQuery processing strategy. But we also expect that XQuery will be extended to support updates, distributed query processing, or new functions and operators. Of course, the overall design of our system must be able to integrate them. We tackle this problem by relying on an algebraic approach that proved successful for relational and object oriented query languages before. This thesis provides evidence that this also holds true for XQuery.

1.4. Contributions

We present an algebraic approach to XQuery processing. The results obtained in this thesis prove that both an algebraic query optimizer and an evaluation engine are well-suited to satisfy the goals outlined above. At the same time, we can leverage the experience gained from building relational and object-oriented databases. However, we cannot reuse these ideas blindly, as the data model of XQuery introduces the following new challenges: (1) It works on trees instead of flat tuples. (2) It is based on sequences of items instead of bags of tuples. (3) It includes a rich type system which is based on XML Schema. Consequently, we have to reconsider fundamental optimizations.

Theory of Algebras over Sequences We formally define the algebraic operators needed to represent XQuery queries in our algebra, NAL. This algebra is defined over sequences of tuples. Thereby, we assure that we capture order semantics of XQuery. We develop a general framework to formally assess the correctness of algebraic equivalences on this algebra. While we still formally prove several specific algebraic equivalences, we introduce properties of algebraic operators that allow us to capture many common algebraic equivalences more succinctly. As another key result, we show that many algebraic equivalences that are valid for algebras over sets or bags do not hold for algebras over sequences.

Translation into Algebra over Sequences We present the translation of a large fragment of XQuery into NAL, our algebra. We prepare this translation step by normalization rewrites. The normalization rewrites transform the query in such a way that it is easy to translate and optimize. Moreover, we rely on an extensible framework to annotate the translated query with type and cardinality information. This information is used in later steps of the optimization process.

Algebraic Rewrites We identify three basic algebraic patterns of nested queries. With a set of algebraic rewrites we are able to remove nesting from these queries. We embed these equivalences into an unnesting strategy; experiments demonstrate the effectiveness and efficiency of our optimizations.

Cost-Based Query Optimization The full power of heuristic rewrites can only be exploited when the optimizer is able to pick efficient implementations for the query. We

motivate the need for a cost-based optimizer to generate optimal plans for queries containing joins or path expressions. We outline the architecture of our cost-based optimizer, in particular how we handle document order.

Discussion of the Design, Implementation, and Experiments We complement our theoretic results with an explanation on the design that underlies the implementation of our XQuery optimizer. We validate its effectiveness in a number of experiments.

1.5. Thesis Outline

The structure of this thesis follows the architecture of our query compiler (see Fig. 1.2). Chapter 2 introduces the logical algebra, NAL, and its algebraic properties. We also survey the most important physical implementations that are available for these algebraic operators. In Chapter 3, we present the translation of a large fragment of XQuery into our algebra. We also motivate the design of our internal query representation and outline how we annotate the query with type and cardinality information. Since the translation of XQuery into our algebra might result in algebraic expressions containing nested query blocks, Chapter 4 introduces our unnesting strategy for XQuery. This chapter extends our previous work [MHM03a, MHM03c, MHM03b, MHM04, MHM06] with remarks on the efficient implementation of our unnesting strategy. Chapter 5, deals with the problem of choosing the most efficient operator order and operator implementation based on cost information. Based on our previous work [MHKM04, MBB$^+$06, MM05a, MM05b], we motivate the need for cost-based optimizations and outline how we solve this problem. Chapter 6 summarizes the results of this thesis and points out future work. The full proofs for the algebraic equivalences proposed in Chapter 2 and 4 can be found in Appendix A.1 and A.2 repectively. Appendix A.3 contains information about the experimental setup used in this thesis.

2. The Natix Algebra

The goal of this thesis is to develop an extensible optimization framework for XQuery. As we want to reason formally about our optimizations, we approach XQuery optimization by translating every query into an algebraic expression. Optimization of XQuery then consists of applying algebraic equivalences to the algebraic expressions. Since we precisely define the semantics of the operators of our algebra, we can prove the correctness of algebraic equivalences. Thereby, we assure that applying them to an algebraic expression does not change its semantics.

In Section 2.1, we present the Natix Logical Algebra (NAL). Before we can define the algebraic operators, we need to arrange for some notation to formally define the operators of the NAL. As this algebra works on sequences of tuples, it preserves duplicates and order. We investigate the properties of our algebra over sequences of tuples in Section 2.2. In Section 2.3, we briefly discuss the Natix Physical Algebra (NPA). The operators available in NPA are sufficient to implement the plans we will discuss in this thesis. In Section 2.4, we discuss work related to the Natix Algebra.

Readers familiar with our algebra may skip this chapter and continue with Chapter 3. In Figure 2.1 we give a brief overview with the formal definitions of the operators in NAL. It might serve as a reference in the remainder of this work.

2.1. The Natix Logical Algebra

Our algebra (NAL) extends the SAL-Algebra [BT99] developed by Beeri and Tzaban. SAL, in turn, is the order-preserving counterpart of the algebra used in [CM93, CM95b]. Both SAL and NAL work on sequences of tuples and allow for nested tuples, i.e. the value of an attribute may be a sequence of tuples.

2.1.1. Notation

Sequences. We denote sequences by $\langle\cdot\rangle$, the empty sequence by ϵ, and sequence concatenation by $\oplus$. Note that sequence concatenation is associative but not commutative. For a sequence e we use $\alpha(e)$ to select its first element and the $\tau(e)$ to retrieve its tail. We equate sequences containing a single item and the item contained. This implicit conversion is demanded by the XQuery specification.

Tuples. Tuples are constructed by using brackets ($[\cdot]$) and concatenated by $\circ$. The set of attributes of a tuple t is denoted by $\mathcal{A}(t)$. The projection of a tuple t on a set of attributes A is denoted by $t|_A$. To access a single attribute B in a tuple, $B \in \mathcal{A}(t)$, we use $t.B$.

For all tuples t_1 and t_2 contained in a sequence of tuples, we demand $\mathcal{A}(t_1) = \mathcal{A}(t_2)$. Given that, we can define the set of attributes $\mathcal{A}(s)$ provided by a sequence s as the set of attributes of the contained tuples. Let e be an expression whose result is a tuple or a sequence of tuples. Then the set of attributes provided in the result of e is denoted by $\mathcal{A}(e)$. For all expressions used in this thesis, it can easily be calculated bottom up.

Binding Attributes. Binding an attribute a of some tuple to a value v is denoted by $[a : v]$. We call an attribute a in an expression e *free* if it occurs in e and is not bound to a value by e. That is, a value for a has to be provided by some other expression, e.g. an outer

Scan Singleton

$$\Box := \langle [] \rangle$$

Selection

$$\sigma_p(e) := \begin{cases} \alpha(e) \oplus \sigma_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \sigma_p(\tau(e)) & \text{else} \end{cases}$$

Tid

$$tid_a(e) := tid_a(e, 1) \quad \text{where}$$
$$tid_a(e, n) := \alpha(e) \circ [a : n] \oplus tid_a(\tau(e), n + 1)$$

Projection

$$\Pi_A(e) := \alpha(e)|_A \oplus \Pi_A(\tau(e))$$

Tid-Duplicate Elimination

$$\Pi_A^{tid_b}(e) := \begin{cases} \alpha(e)|_A \oplus \Pi_A^{tid_b}(\tau(e)) & \text{if } \alpha(e).b \notin \Pi_b(\tau(e)) \\ \Pi_A^{tid_b}(\tau(e)) & \text{else} \end{cases}$$

Map

$$\chi_{a:e_2}(e_1) := \alpha(e_1) \circ [a : e_2(\alpha(e_1))] \oplus \chi_{a:e_2}(\tau(e_1))$$

Product

$$e_1 \overline{\times} e_2 := \begin{cases} \epsilon & \text{if } e_2 = \epsilon \\ (e_1 \circ \alpha(e_2)) \oplus (e_1 \overline{\times} \tau(e_2)) & \text{else} \end{cases}$$
where e_1 is a singleton

Cross Product

$$e_1 \times e_2 := (\alpha(e_1) \overline{\times} e_2) \oplus (\tau(e_1) \times e_2)$$

Join

$$e_1 \bowtie_p e_2 := \sigma_p(e_1 \times e_2)$$

D-Join

$$e_1 < e_2 > := \alpha(e_1) \overline{\times} e_2(\alpha(e_1)) \oplus \tau(e_1) < e_2 >$$

Semijoin

$$e_1 \ltimes_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \ltimes_p e_2) & \text{if } \exists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \ltimes_p e_2 & \text{else} \end{cases}$$

Antijoin

$$e_1 \triangleright_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \triangleright_p e_2) & \text{if } \nexists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \triangleright_p e_2 & \text{else} \end{cases}$$

Left Outer Join

$$e_1 \mathbin{⟕}_p^{g:e} e_2 := \begin{cases} (\alpha(e_1) \bowtie_p e_2) \oplus (\tau(e_1) \mathbin{⟕}_p^{g:e} e_2) & \text{if } (\alpha(e_1) \bowtie_p e_2) \neq \epsilon \\ (\alpha(e_1) \circ \perp_{\mathcal{A}(e_2) \setminus \{g\}} \circ [g : e]) & \text{else} \\ \quad \oplus (\tau(e_1) \mathbin{⟕}_p^{g:e} e_2) & \end{cases}$$

Union

$$e_1 \hat{\cup} e_2 := e_1 \oplus e_2$$

Intersection

$$e_1 \hat{\cap} e_2 := e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2$$

Difference

$$e_1 \hat{-} e_2 := e_1 \triangleright_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2$$

Unnest

$$\mu_{A:g}(e) := (\alpha(e) \times (\Pi_{A:\mathcal{A}(g)}(\alpha(e).g))) \oplus \mu_{A:g}(\tau(e))$$

Unnest Map

$$\Upsilon_{A:e_2}(e_1) := \Pi_{\overline{\hat{a}}}(\mu_{A:\hat{a}}(\chi_{\hat{a}:e_2}(e_1)))$$

Binary Grouping

$$e_1 \Gamma_{g;A_1 \theta A_2;f} e_2 := \alpha(e_1) \circ [g : G(\alpha(e_1))] \oplus (\tau(e_1) \Gamma_{g;A_1 \theta A_2;f} e_2) \quad \text{where}$$
$$G(x) := f(\sigma_{x|_{A_1} \theta A_2}(e_2))$$

Unary Grouping

$$\Gamma_{g;\theta A;f}(e) := \Pi_{A:A'}(\Pi^D_{A':A}(\Pi_A(e)) \Gamma_{g;A' \theta A;f} e)$$

Figure 2.1.: Natix ALgebra: Algebraic Operators

query block. We denote the set of free attributes of an expression e by $\mathcal{F}(e)$. Note that attributes behave the same way as variables: they are bound to a value by some expression and referenced by another one. From now on, we will use the terms variable and attribute interchangeably.

For an expression e_1 possibly containing free variables, and a tuple e_2, we denote by $e_1(e_2)$ the result of evaluating e_1 where bindings of free variables are taken from variable bindings provided by e_2. Of course this requires $\mathcal{F}(e_1) \subseteq \mathcal{A}(e_2)$. For a set of attributes, we define the tuple constructor $\perp_A$ such that it returns a tuple with attributes in A initialized to NULL. Thanks to the NULL-value, we can distinguish empty results from unknown values which is not possible in XQuery yet.

Operations on Sequences. Using these notations, we introduce two elementary operations to construct sequences. The first is $\Box$, which returns a singleton sequence consisting of the empty tuple, i.e. a tuple with no attributes. It is used in order to avoid special cases during the translation of XQuery. The second operation, denoted by $e[a]$, constructs a sequence of tuples with attribute a from a sequence of non-tuple values e. For each value c in e, a tuple is constructed containing a single attribute a whose value is c. More formally, we define $e[a] := \epsilon$ if e is empty, and $e[a] := [a : \alpha(e)] \oplus \tau(e)[a]$ else. We use this operation to map sequences of items in the XQuery data model into sequences of tuples in our data model.

Functions. We refer to an n-ary function, say f, with $f(e_1, \ldots, e_n)$. Sometimes, we will omit the formal parameters in expressions. Then the actual parameters of f must be bound by the enclosing expression. We denote the identity function by *id* and concatenation of functions or operators by $\circ$.

For **result construction** we define a function with signature $\mathcal{C}(type, name, content)$. It constructs a node of the requested node *type*, with given tag *name*, and *content*. We use the arguments $elem$, $attr$, etc. to identify the node type. To support computed constructors, the name and content may reference previously bound variables. Not every argument is meaningful for every node type. But for the sake of simplicity, we ignore this fact. Another proposal to implement result construction with algebraic operators can be found in [FM01].

2.1.2. Operator Definitions

We give the definitions for the order-preserving algebraic operators. For the unordered counterparts see [CM95b]. The NAL algebra allows for nesting of algebraic expressions. For example, within a selection predicate we allow for the occurrence of a nested algebraic expression. Hence, for example, a join within a selection predicate is possible. This simplifies the translation of nested XQuery expressions into the algebra.

We define the algebraic operators recursively on their input sequences. In order to handle the case of empty argument sequences only once and not for every single operator, we arrange the following. For unary operators, if the input sequence is empty, the output sequence is also empty. For binary operators, the output sequence is empty whenever the left operand represents an empty sequence. In the following, let e and e_i be expressions resulting in a sequence of tuples.

General Operators

The order-preserving **selection** operator with predicate p is defined as

$$\sigma_p(e) := \begin{cases} \alpha(e) \oplus \sigma_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \sigma_p(\tau(e)) & \text{else.} \end{cases}$$

We define an auxiliary operator **tid** which numbers the tuples in a sequence by adding an attribute a to each tuple that contains its position within the sequence. We need this

R_1

A_1
3
2
1

R_2

A_2	B
1	2
1	3
2	4
2	5

$R^T :=$

$tid_T(R_1)$

A_1	T
3	1
2	2
1	3

$tid_T(R_1) \bowtie_{A_1=A_2} R_2$

A_1	T	A_2	B
2	2	2	4
2	2	2	5
1	3	1	2
1	3	1	3

$\Pi^{tid_T}(R^T)$

A_1	T	A_2	B
2	2	2	5
1	3	1	3

Figure 2.2.: Example for the tid operator

operator to identify original tuples of a sequence after they have been connected to other tuples, to remember order [MHKM04], or to implement position-aware functions. We define $tid_a(e) := tid_a(e, 1)$ where attribute $a \notin \mathcal{A}(e)$ and

$$tid_a(e, n) := \alpha(e) \circ [a : n] \oplus tid_a(\tau(e), n + 1).$$

For a list of attribute names A, we define the **projection** operator as

$$\Pi_A(e) := \alpha(e)|_A \oplus \Pi_A(\tau(e)).$$

We also define a duplicate-eliminating projection Π^D_A. Besides the projection, its semantics are similar to the `distinct-values` function of XQuery: it does not preserve order. However, we require it to be deterministic and idempotent.

We also need a special order-preserving duplicate-eliminating projection $\Pi^{tid_b}_A$, which removes multiple occurrences of the same tid-value in b that appear in subsequent tuples:

$$\Pi^{tid_b}_A(e) := \begin{cases} \alpha(e)|_A \oplus \Pi^{tid_b}_A(\tau(e)) & \text{if } \alpha(e).B \notin \Pi_B(\tau(e)) \\ \Pi^{tid_b}_A(\tau(e)) & \text{else.} \end{cases}$$

We abbreviate $\Pi^{tid_b}_{\mathcal{A}(e)}(e)$ by $\Pi^{tid_b}(e)$.

Figure 2.2 explains the relationship between the tid operator and the duplicate elimination of tid values. First, we tag each tuple of R_1 with a tid value and bind it to attribute T. Then, we join the resulting tuples with R_2. In this way, some tuples of R_1 find multiple join partners in R_2. Finally, we remove those duplicates using the Π^{tid_T} operator. As a result, we get all tuples of R_1 that have a join partner in R_2. Evidently the content of *some* join partner is still part of the resulting tuples.

Some more variations of projection are useful. If we want to eliminate a set of attributes A, we denote this by $\Pi_{\overline{A}}$. We use Π also for renaming attributes as in $\Pi_{A':A}$. The attributes in the vector A are renamed to those in A'. Attributes other than those mentioned in A remain untouched.

The **map** operator is defined as follows:

$$\chi_{a:e_2}(e_1) := \alpha(e_1) \circ [a : e_2(\alpha(e_1))] \oplus \chi_{a:e_2}(\tau(e_1)).$$

It consumes a sequence of tuples, e_1 and extends a given input tuple $t_1 \in e_1$ by a new attribute $a \notin \mathcal{A}(e)$. The value of this new attribute is computed by evaluating $e_2(t_1)$. Consequently, attribute a might be bound to an item of the XQuery data model or a sequence of tuples.

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

$\chi_{a:\sigma_{A_1=A_2}(R_2)}(R_1) =$

A_1	a
1	$\langle [A_2:1, B:2], [A_2:1, B:3] \rangle$
2	$\langle [A_2:2, B:4], [A_2:2, B:5] \rangle$
3	$\langle\ \rangle$

Figure 2.3.: Example for the map operator

For an example see Figure 2.3. For every tuple in R_1, the map operator collects the tuples of R_2 that match the correlating predicate between A_1 and A_2. These tuples are bound to a sequence-valued attribute a. Note that attribute a of the last tuple in R_1 is bound to an empty sequence because it does not have a matching tuple in R_2.

Join Operators

We define the **cross product** of two tuple sequences as

$$e_1 \times e_2 := (\alpha(e_1) \overline{\times} e_2) \oplus (\tau(e_1) \times e_2)$$

where

$$t_1 \overline{\times} e_2 := \begin{cases} \epsilon & \text{if } e_2 = \epsilon \\ (t_1 \circ \alpha(e_2)) \oplus (t_1 \overline{\times} \tau(e_2)) & \text{else.} \end{cases}$$

Note that t_1 and $\alpha(e_1)$ in this definition represent a single tuple.

We are now prepared to define the **join** operation on ordered sequences:

$$e_1 \bowtie_p e_2 := \sigma_p(e_1 \times e_2)$$

and the **d-join** (or dependent-join, also denoted by $\underrightarrow{\bowtie}$. The arrow points to the right argument whose evaluation dependents on the left argument) as

$$e_1 <e_2> := \alpha(e_1) \overline{\times} e_2(\alpha(e_1)) \oplus \tau(e_1) <e_2> .$$

The d-join as mentioned in [CM93] is similar to the Apply operator [GLJ01] or the MapConcat-operator [RSF06].

We define the **semijoin** as

$$e_1 \ltimes_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \ltimes_p e_2) & \text{if } \exists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \ltimes_p e_2 & \text{else} \end{cases}$$

and the **antijoin** as

$$e_1 \triangleright_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \triangleright_p e_2) & \text{if } \nexists x \in e_2 : p(\alpha(e_1) \circ x) \\ \tau(e_1) \triangleright_p e_2 & \text{else.} \end{cases}$$

The **left outer join**, which will play an essential role in unnesting, is defined as

$$e_1 \mathbin{⟕}_p^{g:e} e_2 := \begin{cases} (\alpha(e_1) \bowtie_p e_2) \oplus (\tau(e_1) \mathbin{⟕}_p^{g:e} e_2) & \text{if } (\alpha(e_1) \bowtie_p e_2) \neq \epsilon \\ (\alpha(e_1) \circ \bot_{\mathcal{A}(e_2) \setminus \{g\}} \circ [g:e]) & \text{else} \\ \quad \oplus (\tau(e_1) \mathbin{⟕}_p^{g:e} e_2) & \end{cases}$$

where $g \in \mathcal{A}(e_2)$. Our definition slightly deviates from the standard left outer join, as we want to use it in conjunction with grouping and (aggregate) functions. Consider, for example, the sequences R_1, R_2, and R_2^{count} in Figure 2.4. Note that R_2^{count} is derived from R_2 by grouping it on A_2 and then counting the tuples in each group. Now we want to join R_1 (via left outer join) with R_2^{count}. Obviously, tuple 3 of R_1 does not have a join

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

$R_2^{count} := \Gamma_{g;=A_2;count}(R_2)$

A_2	g
1	2
2	2

$R_{1,2}^{og} := R_1 \mathbin{⟕}^{g:0}_{A_1=A_2} R_2^{count}$

A_1	A_2	g
1	1	2
2	2	2
3	NULL	0

Figure 2.4.: Example for unary grouping and outer join

partner. The standard left outer join would add a NULL value for g. In our case, having no join partner corresponds to an empty group and the cardinality of it is well-known (0). Hence, we assign this default value to attribute g in sequence $R_{1,2}^{og}$ whenever a tuple of R_1 does not find a join partner. In general, e defines the value given to attribute g for values in e_1 that do not find a join partner in e_2.

Set Operators

The counterparts for set operations on sequences are defined as follows. We define the **union** of two tuple sequences as their concatenation:

$$e_1 \hat{\cup} e_2 := e_1 \oplus e_2.$$

The definition of the **intersection** coincides with the definition of the semijoin:

$$e_1 \hat{\cap} e_2 := e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2.$$

Similarly, the definition of the **difference** is the same as the one for the antijoin:

$$e_1 \hat{-} e_2 := e_1 \triangleright_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2.$$

As for those operations over sets or bags, we demand that both input sequences produce tuples with the same set of attributes, i.e. $\mathcal{A}(e_1) = \mathcal{A}(e_2)$. Note that the second argument in the definition of the intersection and the difference filters tuples of the first argument.

Our definitions differ from those defined in XQuery where all three operators are restricted to sequences of nodes as input. In their result, duplicate nodes must be removed based on node identity. We support this semantic when using the set-based version of those operators. However, when the ordering mode is set to `ordered`, the result nodes must be in document order.

We do not extend the definition of those operators on bags defined in [DGK82] to bags that are order-sensitive. Extending those definitions requires us to decide how we treat positions of tuples in both input sequences. In particular how the order of the result is determined and if equality also includes the (relative) position. Both decisions will be somewhat arbitrary.

Grouping Operators

For the rest of the work let $\theta \in \{=, \leq, \geq, <, >, \neq\}$ be a comparison operator on atomic values. These comparisons will be used in the definition of grouping. More specifically, we will use them to define which items belong to a group. Note that the SQL grouping feature is based only on equality. With nested queries, groups can be formed by applying other comparison operators as well.

As the definitions of the grouping operators are rather involved, we employ the example in Fig. 2.5. Unary grouping (cf. R_2^g in Fig. 2.5) groups R_2 on attribute A_2. The new attribute g is bound to the result of applying function f to all tuples that belong to the same group.

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

$R_2^{count} := \Gamma_{g;=A_2;count}(R_2)$

A_2	g
1	2
2	2

$R_2^g := \Gamma_{g;=A_2;id}(R_2)$

A_2	g
1	$\langle [A_2:1, B:2], [A_2:1, B:3] \rangle$
2	$\langle [A_2:2, B:4], [A_2:2, B:5] \rangle$

$R_{1,2}^g := R_1 \Gamma_{g;A_1=A_2;id} R_2$

A_1	g
1	$\langle [A_2:1, B:2], [A_2:1, B:3] \rangle$
2	$\langle [A_2:2, B:4], [A_2:2, B:5] \rangle$
3	$\langle\ \rangle$

Figure 2.5.: Examples for unary and binary grouping

In the example in Fig. 2.5, attribute g of R_2^g contains a sequence of tuples. They all share the same value on the grouping attribute A_2. For some functions f (in particular aggregate functions), we do not have to keep all the tuples that comprise a group. In our example the values for the count of each group in R_2^{count} can be computed incrementally.

In contrast to unary grouping, which works on one input sequence, binary grouping takes two input sequences as input (cf. $R_{1,2}^g$ in Fig. 2.5). Each tuple of the left input R_1 defines a group. The tuples of the right input R_2 are matched to these groups based on the predicate $A_1 \theta A_2$. The vector of attributes in A_1 and A_2 must have equal size. When they contain more than one attribute, the result of the comparison is the conjunction of comparisons between pairs of attributes at the same position in the vaectors. If the predicate evaluates to true, the tuple of R_2 belongs to the group under consideration. Note that each tuple of R_2 can belong to multiple groups. Again, function f is used to combine the tuples in each group. For the identity function id, the result is a sequence of tuples. In this case, binary grouping is identical to the nestjoin [SABdB94]. Note that in Fig. 2.5, the last group does not find matching tuples in R_2. Therefore, this group contains an empty sequence. This is important when we access the sequence-valued attribute g. Also notice that binary grouping computes the same result as the map operator in Fig. 2.3. For this reason, the binary grouping operator is used for unnesting nested queries or in OLAP queries [CM93, SABdB94, CKMP97, ACJK01, MHM06].

We define unary grouping in terms of binary grouping. In our definitions we have $A_i \subseteq \mathcal{A}(e_i)$ and $g \notin (A_1 \cup A_2)$. The new attribute g is bound to the result of applying function f to all tuples that belong to the same group. Hence, we start with the formal definition of **binary grouping**:

$$e_1 \Gamma_{g;A_1 \theta A_2;f} e_2 := \alpha(e_1) \circ [g : G(\alpha(e_1))] \ \oplus\ (\tau(e_1) \Gamma_{g;A_1 \theta A_2;f} e_2).$$

where for a function f we define $G(x) := f(\sigma_{x|_{A_1} \theta A_2}(e_2))$. Now, **unary grouping** can be formally defined as follows:

$$\Gamma_{g;\theta A;f}(e) := \Pi_{A:A'}(\Pi_{A':A}(\Pi_A^D(e)) \Gamma_{g;A' \theta A;f} e).$$

XPath Evaluation

We subsume the following operators as operators for XPath evaluation because in this thesis they are predominantly used for this purpose. However, their definitions suggest that these operators are not restricted to XPath evaluation.

The **unnest** operator gets a sequence of tuples containing an attribute g as argument. Attribute g is bound to a sequence of tuples. The unnest operator unnests g by producing

a result tuple for every tuple contained in g. Every result tuple contains the concatenation of the attributes in the argument e and a tuple in the sequence-valued attribute g. The order-preserving analogon to the well-known unnest operator is defined as

$$\mu_{A:g}(e) := (\alpha(e) \times (\Pi_{A:\mathcal{A}(g)}(\alpha(e).g))) \oplus \mu_{A:g}(\tau(e))$$

where A is a vector of attributes, $A \cap \mathcal{A}(e) = \emptyset$, and $\alpha(e).g$ retrieves the sequence of tuples stored in attribute g. The unnest operator creates a new tuple for each tuple in the sequence bound to $\alpha(e).g$. This new tuple contains the attributes and bindings of $\alpha(e)$ and the values of one tuple in $\alpha(e).g$. Thereby, the vector of attributes in g is renamed to the vector of attribute names given in A.

In some rare cases, operators following the unnest operator refer to the attribute g. Hence, the unnest operator preserves the sequence-valued attribute g. However, in most cases we will ignore its existence. But, we may use the fact that for the sequences R_2 and R_2^g in Fig. 2.5 it holds that $R_2 = \Pi_{A_2:A_3}(\Pi_{A_3B}(\mu_{\mathcal{A}(g):g}(\Pi_{A_3:A_2}(R_2^g))))$. Hence, the unnest operator can extract the sequence-valued attributes computed by a grouping operation.

As a very convenient abbreviation, we define the **unnest map** operator as follows:

$$\Upsilon_{A:e_2}(e_1) := \Pi_{\overline{\hat{a}}}(\mu_{A:\hat{a}}(\chi_{\hat{a}:e_2}(e_1)))$$

It first materializes a sequence of tuples in a new sequence-valued attribute $\hat{a}$ which is then immediately unnested. As a result, the tuples of e_1 are extended by the attributes in e_2, which are renamed to the vector of attribute names in A. Basically, the unnest map operator has the same semantics as our d-join (see Eqv. 2.17 in Section 2.2.3).

We mainly use the unnest map operator to evaluate XPath location steps. Therefore, we translate the XPath expressions after normalization as presented in [BKHM05]. Note that our translation of XPath expressions yields sequences of tuples as opposed to sequences of items as defined in XPath [DFF+07]. For path expressions the final projection establishes the bindings required by the expression that embeds the path expression.

2.2. Algebraic Equivalences

After a query is translated into an algebraic expression it is likely that it can be improved further in two ways. First, efficient implementations of the algebraic operators in NAL can lead to substantial improvements in execution time. We discuss possible implementations of our algebra in the next section. Second, algebraic equivalences that transform an algebraic expression into a different but equivalent algebraic expression give the query optimizer the freedom to find more efficient query evaluation plans. In this section we discuss fundamental algebraic equivalences that hold for our algebra over sequences.

While the relational algebra is based on sets, query languages such as SQL work on bags (also known as multisets). Both sets and bags have been studied intensively because algebraic equivalences for sets or bags are at the core of most query optimizers today. As a new challenge, an algebra over sequences is sensitive to both duplicates and order of tuples. Hence, some equivalences known for algebras over sets or bags are no longer valid. Most importantly cross products – and consequently joins – are not commutative any more. Thus, when we commute the arguments of a join, we have to repair order afterwards.

We do not simply enumerate algebraic equivalences that hold for our algebra. Instead, we first discuss associativity and commutativity for the binary operators in our algebra. Then we apply the notion of *linearity* [vB90, CM95a] to our algebra over sequences. Linearity is an important property that allows us to reorder operators. Hence, we need not check every pair of algebraic operators when we consider their reorderability. Finally, we give additional algebraic equivalences that do not follow immediately from linearity or that hold for non-linear algebraic operators. In Appendix A.1 we present the proofs of all equivalences presented in this section.

$$e_1 \times (e_2 \times e_3) = (e_1 \times e_2) \times e_3 \quad (2.1)$$
$$e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) = (e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3 \quad (2.2)$$
$$e_1 \bowtie\!\!\!\!\bowtie_{p_1} (e_2 \bowtie\!\!\!\!\bowtie_{p_2} e_3) = (e_1 \bowtie\!\!\!\!\bowtie_{p_1} e_2) \bowtie\!\!\!\!\bowtie_{p_2} e_3 \quad (2.3)$$
$$e_1 \hat{\cup} (e_2 \hat{\cup} e_3) = (e_1 \hat{\cup} e_2) \hat{\cup} e_3 \quad (2.4)$$
$$e_1 \hat{\cap} (e_2 \hat{\cap} e_3) = (e_1 \hat{\cap} e_2) \hat{\cap} e_3 \quad (2.5)$$

Figure 2.6.: Associativity of binary operators in NAL

2.2.1. Commutativity and Associativity

Let us begin with commutativity. As we will see below, none of the binary operators is commutative for sequences. This is a substantial restriction compared to algebras over bags or sets.

Cross product is not commutative as a counter example consider (see [Moe03]):

$$\begin{aligned} e_1 &= <[a:1],[a:2]> \\ e_2 &= <[b:1],[b:2]> \\ e_1 \times e_2 &= <[a:1,b:1],[a:1,b:2],[a:2,b:1],[a:2,b:2]> \\ e_2 \times e_1 &= <[a:1,b:1],[a:2,b:1],[a:1,b:2],[a:2,b:2]>. \end{aligned}$$

Join is not commutative which is a direct consequence of this property for the cross product.

Semijoin, antijoin, d-join, outer join, and binary grouping are not commutative for obvious reasons.

Union is not commutative because

$$\begin{aligned} e_1 &= <[a:1],[a:2]> \\ e_2 &= <[a:3],[a:4]> \\ e_1 \hat{\cup} e_2 &= <[a:1],[a:2],[a:3],[a:4]> \\ e_2 \hat{\cup} e_1 &= <[a:3],[a:4],[a:1],[a:2]>. \end{aligned}$$

Intersection is not commutative because it filters tuples of the left input based on the tuples available in the right input. This asymmetry breaks the commutativity as the following example shows:

$$\begin{aligned} e_1 &= <[a:1],[a:2]> \\ e_2 &= <[a:2],[a:2]> \\ e_1 \hat{\cap} e_2 &= <[a:2]> \\ e_2 \hat{\cap} e_1 &= <[a:2],[a:2]>. \end{aligned}$$

Difference is not commutative because it is not even commutative for sets.

We resume with checking the associativity of binary algebraic operators. Figure 2.6 summarizes our results.

Cross product is associative (Eqv. 2.1)

Preconditions None.

Basic Idea It does not matter how we parenthesize cross products. This is an important property for ordering cross products in the plan generator.

Join is associative (Eqv. 2.2)

Preconditions $\mathcal{F}(p_1) \subset \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$ and $\mathcal{F}(p_2) \subset \mathcal{A}(e_2) \cup \mathcal{A}(e_3)$

Basic Idea Associativity of joins is a direct consequence of associativity of cross products. Note that the preconditions can be relaxed. But then associativity might introduce cross products and the predicates involved cannot simply be copied. Join order optimization with associative (but not commutative) joins is investigated by Moerkotte [Moe03].

Semijoin and Antijoin are not associative. Consider the algebraic expression $(e_1 \ltimes_p e_2) \ltimes_q e_3$. In this expression the semijoin with e_3 as right argument has access to the attributes of e_1 but not those of e_2. On the other hand, in expression $e_1 \ltimes_p (e_2 \ltimes_q e_3)$ the semijoin with e_3 as right argument has access to the attributes of e_2 but not to those of e_1. For the same reasons, the antijoin operator is not associative.

Outer join is not generally associative. Galindo-Legaria et al. [RGL90, GLR97] observed that outer joins are not associative in general and survey algebraic rewrites that are valid or invalid for outer joins in an algebra over sets. As a consequence, outer joins over sequences of tuples are neither associative nor commutative in general. In Figure 2.6, we give an equivalence involving a left outer join. Later, we will discuss several more equivalences.

Outer join associativity (Eqv. 2.3)

Preconditions $\mathcal{F}(p_1) \subset \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$, $\mathcal{F}(p_2) \subset \mathcal{A}(e_2) \cup \mathcal{A}(e_3)$, neither the result of p_1 nor p_2 depend on the position, and p_2 must be strong w.r.t e_2.

Basic Idea The requirement that p_2 is strong with respect to expression e_2 means that p_2 evaluates to false if all attributes in e_2 contain $\perp_{\mathcal{A}(e_2)}$. Given these constraints above, left outer joins can be reordered,

Union is associative (Eqv. 2.4)

Preconditions $\mathcal{A}(e_1) = \mathcal{A}(e_2) = \mathcal{A}(e_3)$.

Basic Idea Associativity of the union operator directly follows from associativity of sequence concatenation.

Intersection is associative (Eqv. 2.5)

Preconditions $\mathcal{A}(e_1) = \mathcal{A}(e_2) = \mathcal{A}(e_3)$.

Basic Idea Associativity of the intersection operator results from the transitivity of the equality comparison.

Difference is not associative The difference operator is not even associative for sets as the example below shows:

$$\begin{aligned} e_1 &= <[a:1]> \\ e_2 &= <[a:1]> \\ e_3 &= <[a:1]> \\ e_1 \hat{-} (e_2 \hat{-} e_3) &= <[a:1]> \\ (e_1 \hat{-} e_2) \hat{-} e_3 &= \epsilon. \end{aligned}$$

Binary Grouping is not associative The binary grouping operator is not even associative for sets. Consider the expression $(e_1 \Gamma_{g_1;A_1\theta A_{2_1};f_1} e_2)\Gamma_{g_2;A_{2_2}\theta A_3;f_2} e_3$. This expression produces tuples containing attributes $\mathcal{A}(e_1) \cup g_1 \cup g_2$. In contrast, expression $e_1 \Gamma_{g_1;A_1\theta A_{2_1};f_1}(e_2 \Gamma_{g_2;A_{2_2}\theta A_3;f_2} e_3)$ produces tuples containing attributes $\mathcal{A}(e_1) \cup g_1$.

In Section 2.2.3, we discuss algebraic equivalences for the outer join and binary grouping that still hold.

2.2.2. Linearity

Finding the optimal plan for a query relies on the possibility to reorder the algebraic operators in an algebraic expression. Many queries contain different unary and binary operators such as join, outer join, grouping, and selection. Thus, we have to investigate the reorderability of all pairs of algebraic operators to be able to exploit all possibilities. For n algebraic operators this means that we have to formally prove reorderability of n^2 pairs of operators.

In Section 2.2.1, we have only looked at binary operators in isolation. For the general case we still do not have to provide full proofs for all n^2 combinations because we apply the notion of *linearity* to our algebra over sequences [vB90, CM95a]. Based on the linearity we can state conditions when two operators are reorderable. This results in a more extensible and concise test.

Definition 1 (Linearity) *Let f be a unary operator that consumes an input sequence s and returns a result sequence s'. Then $f : s \rightarrow s'$ is called* linear*, iff*

1. $f(\epsilon) = \epsilon$
2. $f(s_1 \oplus s_2) = f(s_1) \oplus f(s_2)$ *where* $s = s_1 \oplus s_2$.

Intuitively, a unary operator is linear if applying the operator to subsequences and concatenating the results of these applications does not change the overall result.

We can generalize the notion of linearity to n-ary operators as follows. Let f be an n-ary operator over sequences $s_1, s_2, \ldots, s_i, \ldots, s_n$, then f is linear in its i-th argument iff

1. $f(s_1, \ldots, \epsilon, \ldots, s_n) = \epsilon$
2. $f(s_1, \ldots, s_{i_1} \oplus s_{i_2}, \ldots, s_n) = f(s_1, \ldots, s_{i_1}, \ldots, s_n) \oplus f(s_1, \ldots, s_{i_2}, \ldots, s_n)$ where $s_i = s_{i_1} \oplus s_{i_2}$.

Since we want to investigate how several algebraic operators interact, let us note that the application of linear operators in a sequence is again a linear operation.

Corollary 1 *Let f and g be linear operators over some sequence $s = s_1 \oplus s_2$. Then their concatenation $f \circ g$ is again a linear operation, i.e.*

$$(f \circ g)(s_1 \oplus s_2) \quad = \quad (f \circ g)(s_1) \oplus (f \circ g)(s_2).$$

Proof: For the first condition we have:

$$\begin{aligned}(f \circ g)(\epsilon) &= \epsilon \\ &= (f \circ g)(\epsilon) \oplus (f \circ g)(\epsilon).\end{aligned}$$

Now, we show the second condition:

$$\begin{aligned}(f \circ g)(s_1 \oplus s_2) &= f(g(s_1 \oplus s_2)) \\ &= f(g(s_1) \oplus g(s_2)) \\ &= f(g(s_1)) \oplus f(g(s_2)) \\ &= (f \circ g)(s_1) \oplus (f \circ g)(s_2).\end{aligned}$$

□

We will make use of Corollary 1 when we discuss the reorderability of arbitrary sequences of linear operators in the following section. But we also use this corollary to (dis-) prove the linearity of operators that can be expressed in terms of other algebraic operators.

Notice that we have to show the linearity of every operator explicitly. Therefore, we provide inductive proofs. For non-linear operators we give counter examples.

To see why we cannot simplify the proofs to the concatenation of singleton sequences, consider an algebraic operator $\mathcal{RM}3$ that removes the third tuple. Let t_1, t_2, t_3 be tuples. Then, operator $\mathcal{RM}3$ clearly is not linear because

$$\begin{aligned} \mathcal{RM}3(< t_1, t_2, t_3 >) &= < t_1, t_2 > \quad \text{but} \\ \mathcal{RM}3(< t_1, t_2 >) \oplus \mathcal{RM}3(< t_3 >) &= < t_1, t_2, t_3 >, \quad \text{i.e.} \end{aligned}$$

this operator violates the second condition of Definition 1. However, by considering only singleton sequences, we do not detect this violation:

$$\begin{aligned} \mathcal{RM}3(< t_1 > \oplus < t_2 >) &= < t_1, t_2 > \\ &= \mathcal{RM}3(< t_1 >) \oplus \mathcal{RM}3(< t_2 >). \end{aligned}$$

Similar examples can be constructed to show that we cannot shortcut this test for bags or sets either.

Therefore, we examine the linearity of every operator in NAL; Figure 2.7 summarizes these results. To show that an algebraic operator is not linear in some argument, we present a counter example. We prove the linearity of the operators by induction over the length of input sequence s. In all these proofs t denotes a singleton sequence.

Notice that the base case follows directly from the first condition in Definition 1. To see this, consider some operator f for which we have verified that $f(\epsilon) = \epsilon$. Then we can show that $f(\epsilon \oplus \epsilon) = f(\epsilon) = f(\epsilon) \oplus \epsilon = f(\epsilon) \oplus f(\epsilon)$.

To avoid clutter, we will only present a simplified inductive step of each proof. We need to verify the second condition in Definition 1 for arbitrary sequences. However, the following corollary allows us to restrict ourselfs to the concatenation of a singleton sequence, t, and sequence of arbitrary length, s.

Corollary 2 *Let $s = (s_1 \oplus s_2)$ be an arbitrary non-empty sequence and f be a unary operator. If $f(\alpha(s) \oplus \tau(s)) = f(\alpha(s)) \oplus f(\tau(s))$ holds then also $f(s_1 \oplus s_2) = f(s_1) \oplus f(s_2)$.*

Proof: We can prove this corollary by induction over the length of sequence s_1. We use that sequence concatenation is associative.

Base Case : $|s_1| = 1$: then $s_1 = \alpha(s)$ and $s_2 = \tau(s)$ and the claim follows from the prerequisite.

Inductive Hypothesis : $f(\alpha(s) \oplus \tau(s)) = f(\alpha(s)) \oplus f(\tau(s)) \Rightarrow f(s_1 \oplus s_2) = f(s_1) \oplus f(s_2)$ holds for $|s_1| > 0$.

Inductive Step : $(|s_1| - 1) \rightarrow |s_1|$

$$\begin{aligned} f(s_1) \oplus f(s_2) &= f(\alpha(s_1) \oplus \tau(s_1)) \oplus f(s_2) \\ &= (f(\alpha(s_1)) \oplus f(\tau(s_1))) \oplus f(s_2) \\ &= f(\alpha(s_1)) \oplus (f(\tau(s_1)) \oplus f(s_2)) \\ &\overset{IH}{=} f(\alpha(s_1)) \oplus (f(\tau(s_1) \oplus s_2)) \\ &= f(s_1 \oplus s_2) \end{aligned}$$

□

We continue with the proofs of the operators in NAL.

Scan Singleton Since this operator does not have any argument, linearity does not make sense for this operator.

Selection is linear because

1. $\sigma_p(\epsilon) = \epsilon$ and
2. **case 1:** $p(t)$ = true, then $\sigma_p(t \oplus s) = t \oplus \sigma_p(s) = \sigma_p(t) \oplus \sigma_p(s)$
   case 2: $p(t)$ = false, then $\sigma_p(t \oplus s) = \epsilon \oplus \sigma_p(s) = \sigma_p(t) \oplus \sigma_p(s)$.

Tid is not linear. Consider the following counter example:

$$\begin{aligned} tid_t(< [a:1] > \oplus < [a:2] >) &= < [a:1,t:1],[a:2,t:2] > \quad \text{but} \\ tid_t(< [a:1] >) \oplus tid_t(< [a:2] >) &= < [a:1,t:1],[a:2,t:1] > . \end{aligned}$$

Tid-Duplicate Elimination is not linear. Consider the following counter example:

$$\begin{aligned} \Pi^{tid_t}(< [a:1,t:1],[a:2,t:1] >) &= < [a:2,t:1] > \quad \text{but} \\ \Pi^{tid_t}(< [a:1,t:1] >) \oplus \Pi^{tid_t}(< [a:2,t:1] >) &= < [a:1,t:1],[a:2,t:1] > . \end{aligned}$$

Duplicate Elimination is not linear. Consider the following counter example:

$$\begin{aligned} \Pi^D_a(< [a:1] > \oplus < [a:1] >) &= < [a:1] > \quad \text{but} \\ \Pi^D_a(< [a:1] >) \oplus \Pi^D_a(< [a:1] >) &= < [a:1],[a:1] > . \end{aligned}$$

Projection is linear because

1. $\Pi_A(\epsilon) = \epsilon$ and
2. $\Pi_A(t \oplus s) = t|_A \oplus \Pi_A(s) = \Pi_A(t) \oplus \Pi_A(s)$.

Map is linear because

1. $\chi_{a:e_2}(\epsilon) = \epsilon$ and
2. $\chi_{a:e_2}(t \oplus s) = t \circ [a : e_2(t)] \oplus \chi_{a:e_2}(s) = \chi_{a:e_2}(t) \oplus \chi_{a:e_2}(s)$.

Product is linear in its second argument Note that the first argument of the Product, e_1, must be a singleton sequence. Hence, linearity in this argument is not relevant.

1. $e_1 \overline{\times} \epsilon = \epsilon$ and
2. $e_1 \overline{\times}(t \oplus s) = (e_1 \circ t) \oplus (e_1 \overline{\times} s) = (e_1 \overline{\times} t) \oplus (e_1 \overline{\times} s)$.

Cross Product is ...

... **linear in its first argument**

1. $\epsilon \times e_2 = \epsilon$ and
2. $(t \oplus s) \times e_2 = (t \overline{\times} e_2) \oplus (s \times e_2) = (t \times e_2) \oplus (s \times e_2)$.

... **not linear in its second argument.** Consider the following counter example:

$$\begin{aligned} & < [a:1],[a:2] > \times (< [b:1] > \oplus < [b:2] >) \\ = \; & < [a:1,b:1],[a:1,b:2],[a:2,b:1],[a:2,b:2] > \end{aligned}$$

but

$$\begin{aligned} & (< [a:1],[a:2] > \times < [b:1] >) \oplus (< [a:1],[a:2] > \times < [b:2] >) \\ = \; & < [a:1,b:1],[a:2,b:1],[a:1,b:2],[a:2,b:2] > . \end{aligned}$$

Join is linear in its first argument Linearity in the first argument follows from linearity of σ, $\times$ and Corollary 1. Since $\times$ is not linear in its second argument, $\bowtie$ cannot be linear in its second argument either (e.g. take $p = true$).

D-Join is linear in its first argument because

1. $\epsilon <e_2>= \epsilon$ and
2. $(t \oplus s) <e_2>= (t \overline{\times} e_2(t)) \oplus (s <e_2>) = (t <e_2>) \oplus (s <e_2>)$.

Note that linearity of the second argument is not meaningful for the d-join because the result of evaluating the second argument depends on the values computed in the first argument.

Semijoin is ...

linear in its first argument because

1. $\epsilon \ltimes_p e_2 = \epsilon$ and
2. **case 1:** $\exists x \in e_2 : p(t \circ x)$ then
   $(t \oplus s) \ltimes_p e_2 = t \oplus (s \ltimes_p e_2) = (t \ltimes_p e_2) \oplus (s \ltimes_p e_2)$
   case 2: $\neg \exists x \in e_2 : p(t \circ x)$ then
   $(t \oplus s) \ltimes_p e_2 = \epsilon \oplus (s \ltimes_p e_2) = (t \ltimes_p e_2) \oplus (s \ltimes_p e_2)$.

not linear in its second argument. Consider the following counter example:

$$\begin{aligned} e_1 &= < [a:1],[a:1] > \\ e_2 &= < [b:1],[b:1] > \\ e_1 \ltimes_{a=b} e_2 &= e_1 \quad \text{but} \\ (e_1 \ltimes_{a=b} < [b:1] >) \oplus (e_1 \ltimes_{a=b} < [b:1] >) &= < [a:1],[a:1],[a:1],[a:1] > . \end{aligned}$$

Antijoin is ...

linear in its first argument because

1. $\epsilon \triangleright_p e_2 = \epsilon$ and
2. **case 1:** $\exists x \in e_2 : p(t \circ x)$ then
   $(t \oplus s) \triangleright_p e_2 = \epsilon \oplus (s \triangleright_p e_2) = (t \triangleright_p e_2) \oplus (s \triangleright_p e_2)$
   case 2: $\neg \exists x \in e_2 : p(t \circ x)$ then
   $(t \oplus s) \triangleright_p e_2 = t \oplus (s \triangleright_p e_2) = (t \triangleright_p e_2) \oplus (s \triangleright_p e_2)$.

not linear in its second argument. Consider the following counter example:

$$\begin{aligned} e_1 &= < [a:1],[a:1] > \\ e_2 &= < [b:2] > \\ e_1 \triangleright_{a=b} e_2 &= e_1 \quad \text{but} \\ (e_1 \triangleright_{a=b} < [b:2] >) \oplus (e_1 \triangleright_{a=b} < [b:2] >) &= < [a:1],[a:1],[a:1],[a:1] > . \end{aligned}$$

Left Outer Join is ...

linear in its first argument because

1. $\epsilon \ \mathbb{N}_p^{g:e} \ e_2 = \epsilon$ and
2. **case 1:** $t \bowtie_p e_2 \neq \epsilon$ then linearity follows from linearity of $\bowtie$
   case 2: $t \bowtie_p e_2 = \epsilon$ then

$$\begin{aligned}(t \oplus s) \ \mathbb{N}_p^{g:e} \ e_2 &= (t \circ \perp_{\mathcal{A}(e_2)\setminus\{g\}} \circ [g:e]) \oplus (s \ \mathbb{N}_p^{g:e} \ e_2) \\ &= (t \ \mathbb{N}_p^{g:e} \ e_2) \oplus (s \ \mathbb{N}_p^{g:e} \ e_2).\end{aligned}$$

not linear in its second argument because $e_1 \ \mathbb{N}_p^{g:e} \ \epsilon = \epsilon$ holds only if $e_1 = \epsilon$.

Union is not linear

in its first argument. Consider the following counter example:

$$\begin{aligned}(<[a:1]> \oplus <[a:2]>) \ \hat{\cup} \ <[a:3]> &= <[a:1],[a:2],[a:3]> \quad \text{but} \\ (<[a:1]> \ \hat{\cup} \ <[a:3]>) \oplus & \\ (<[a:2]> \ \hat{\cup} \ <[a:3]>) &= <[a:1],[a:3],[a:2],[a:3]>.\end{aligned}$$

in its second argument. Consider the following counter example:

$$\begin{aligned}<[a:1]> \ \hat{\cup} \ (<[a:2]> \oplus <[a:3]>) &= <[a:1],[a:2],[a:3]> \quad \text{but} \\ (<[a:1]> \ \hat{\cup} \ <[a:2]>) \oplus & \\ (<[a:1]> \ \hat{\cup} \ <[a:3]>) &= <[a:1],[a:2],[a:1],[a:3]>.\end{aligned}$$

The counter examples show that union is not even linear for bags.

Intersection is ...

linear in its first argument. This follows directly from the definition of the intersection operator in terms of the semijoin and linearity and the semijoin in its first argument.

not linear in its second argument. Consider the following counter example:

$$\begin{aligned}e_1 &= <[a:1],[a:2]> \\ e_2 &= <[a:2],[a:1]> \\ e_1 \ \hat{\cap} \ e_2 &= e_1 \quad \text{but} \\ (e_1 \ \hat{\cap} \ <[a:2]>) \oplus (e_1 \ \hat{\cap} \ <[a:1]>) &= e_2.\end{aligned}$$

Difference is ...

linear in its first argument. This follows directly from the definition of the difference operator in terms of the antijoin and linearity and the antijoin in its first argument.

not linear in its second argument. Consider the following counter example:

$$\begin{aligned}e_1 &= <[a:1],[a:2]> \\ e_2 &= <[a:2],[a:1]> \\ e_1 \ \hat{-} \ e_2 &= \epsilon \quad \text{but} \\ (e_1 \ \hat{-} \ <[a:2]>) \oplus (e_1 \ \hat{-} \ <[a:1]>) &= e_1.\end{aligned}$$

Unnest is linear because

1. $\mu_{A:g}(\epsilon) = \epsilon$ and
2. $\mu_{A:g}(t \oplus s) = (t \times (t.g)_{|A:\mathcal{A}(g)}) \oplus \mu_{A:g}(s) = \mu_{A:g}(t) \oplus \mu_{A:g}(s)$.

Unnest Map is linear by linearity of μ and χ and Corollary 1.

Binary Grouping is ...

... **linear in its first argument** because

1. $\epsilon\ \Gamma_{g;A_1\theta A_2;f} e_2 = \epsilon$ and

$$\begin{aligned}(t \oplus s)\Gamma_{g;A_1\theta A_2;f} e_2 &= (t \circ [g : G(t)]) \oplus (s\ \Gamma_{g;A_1\theta A_2;f} e_2) \\ &= (t\ \Gamma_{g;A_1\theta A_2;f} e_2) \oplus (s\ \Gamma_{g;A_1\theta A_2;f} e_2).\end{aligned}$$

... **not linear in its second argument.** Consider the following counter example:

$$\begin{aligned}<[a:1]> \Gamma_{g;a=b;count}(<[b:1]> \oplus <[b:1]>) &= <[a:1,g:2]> \quad \text{but} \\ (<[a:1]> \Gamma_{g;a=b;count} <[b:1]>) \oplus & \\ (<[a:1]> \Gamma_{g;a=b;count} <[b:1]>) &= <[a:1,g:1],[a:1,g:1]>.\end{aligned}$$

Unary Grouping is not linear. Consider the following counter example:

$$\begin{aligned}\Gamma_{g;=a;count}(<[a:1]> \oplus <[a:1]>) &= <[a:1,g:2]> \quad \text{but} \\ (\Gamma_{g;=a;count}(<[a:1]>)) \oplus (\Gamma_{g;=a;count}(<[a:1]>)) &= <[a:1,g:1],[a:1,g:1]>.\end{aligned}$$

In Figure 2.7, we summarize the results of our discussion on the linearity of operators in NAL. Many operators are linear. In particular, all join operators are linear in their first argument. Linearity of these operators allows us to discuss reordering algebraic operators in a more general fashion. However, neither the Cross product nor the the Join is linear in its second argument.

2.2.3. Reorderability

Now we will exploit linearity to decide if two algebraic operators can be reordered. The following definition states the conditions that must hold for two operators to be reorderable.

Definition 2 (Reorderability) *Let f and g be unary operators that map sequence r (s) to sequence r' (s'), i.e. $f : r \to r'$ and $g : s \to s'$, s an arbitrary sequence, and t_1, t_2 arbitrary singleton sequences. If for f and g the two conditions*

1. *f and g are linear, and*
2. *$f(g(s)) = g(f(s)) \Leftrightarrow f(g(t_1 \oplus t_2)) = g(f(t_1)) \oplus g(f(t_2))$.*

hold then f and g are reorderable.

The second condition in Definition 2 assures that the operators do not interfere in their producer-consumer relationship. Notice that algebraic expressions in XPath or XQuery might depend on the order of tuples in a sequence. Thus, now in contrast to data models based on sets or bags the order of the resulting sequence matters.

The second condition also states that we can simplify the test of reorderability to singleton sequences. This is valid because we can construct a sequence of arbitrary length by concatenating it from singleton sequences. In addition, we have shown in Section 2.2.2 the linearity of the involved operators for sequences of arbitrary length, and reordering them does not invalidate this property. We now prove the second condition of Definition 2 inductively.

Proof:

Operator	first argument (e_1)	second argument (e_2)
Scan Singleton	-	-
Selection	$\surd$	-
Tid	x	-
Projection	$\surd$	-
Tid-Duplicate Elimination	x	-
Map	$\surd$	-
Product	-	$\surd$
Cross Product	$\surd$	x
Join	$\surd$	x
D-Join	$\surd$	x
Semijoin	$\surd$	x
Antijoin	$\surd$	x
Left Outer Join	$\surd$	x
Union	x	x
Intersection	$\surd$	x
Difference	$\surd$	x
Unnest	$\surd$	-
Unnest Map	$\surd$	-
Binary Grouping	$\surd$	x
Unary Grouping	x	-

$\surd$: is linear; x: is not linear; -: is not applicable

Figure 2.7.: Linearity of operators in NAL

"⇒": This direction is trivially true. We simply define $s = t_1 \oplus t_2$ and use that f and g are linear.

"⇐": Let s be a sequence and t_i tuples (singleton sequences resp.).

Base Case We have to consider the empty sequence and the singleton sequence. (1) The case of the empty sequence, i.e. $s = \epsilon$, reduces to the first case in Definition 1. (2) The case of the singleton sequence, i.e. $s = t_1$, follows directly from the prerequisites, i.e. $f(g(s)) = f(g(t_1 \oplus \epsilon)) = g(f(t_1)) \oplus \epsilon = g(f(s))$.

Inductive Hypothesis $f(g(s)) = g(f(s)) \Leftrightarrow f(g(t_1 \oplus t_2)) = g(f(t_1)) \oplus g(f(t_2))$ for $|s| > 0$.

Inductive Step $s \rightarrow t_2 \oplus s$.

Since we have a non-empty sequence, we have a sequence $t_2 \oplus s$.

$$
\begin{aligned}
& f(g(t_1 \oplus (t_2 \oplus s))) \\
= \;& f(g(t_1 \oplus (t_2 \oplus s))) \\
= \;& f(g((t_1 \oplus t_2) \oplus s)) \\
= \;& f(g(t_1 \oplus t_2)) \oplus f(g(s)) \\
\overset{PR}{=} \;& (g(f(t_1)) \oplus g(f(t_2))) \oplus f(g(s)) \\
\overset{IH}{=} \;& (g(f(t_1)) \oplus g(f(t_2))) \oplus g(f(s)) \\
= \;& g(f(t_1)) \oplus (g(f(t_2)) \oplus g(f(s))) \\
= \;& g(f(t_1)) \oplus g(f(t_2 \oplus s)) \\
= \;& g(f(t_1)) \oplus g(f(t_2 \oplus s)) \\
= \;& g(f(t_1 \oplus (t_2 \oplus s)))
\end{aligned}
$$

$$\sigma_{p_1}(\sigma_{p_2}(e_1)) = \sigma_{p_2}(\sigma_{p_1}(e_1)) \quad (2.6)$$
$$\sigma_{p_1}(e_1 \times e_2) = \sigma_{p_1}(e_1) \times e_2 \quad (2.7)$$
$$\sigma_{p_1}(e_1 \bowtie_p e_2) = \sigma_{p_1}(e_1) \bowtie_p e_2 \quad (2.8)$$
$$\sigma_{p_1}(e_1 \ltimes_p e_2) = \sigma_{p_1}(e_1) \ltimes_p e_2 \quad (2.9)$$
$$\sigma_{p_1}(e_1 \rhd_p e_2) = \sigma_{p_1}(e_1) \rhd_p e_2 \quad (2.10)$$
$$\sigma_{p_1}(e_1 \; \mathbb{N}_p^{g:e} \; e_2) = \sigma_{p_1}(e_1) \; \mathbb{N}_p^{g:e} \; e_2 \quad (2.11)$$
$$\sigma_p(e_1 \Gamma_{g;A_1\theta A_2;f} e_2) = (\sigma_p(e_1))\Gamma_{g;A_1\theta A_2;f} e_2 \quad (2.12)$$

Figure 2.8.: Reorderability of operators in NAL (examples)

In the step marked with PR, we exploit the prerequisite that we can reorder f and g on singleton sequences. The remaining steps either rely on the linearity of f and g or on the associativity of $\oplus$.

□

Definition 2 directly extends to n-ary operators. As a result, a linear operator can be pushed into or pulled out of the i-th argument of an n-ary operator if this operator is linear in its i-th argument and the second condition mentioned in Definition 2 holds.

Definition 2 simplifies matters because checking the conditions on the involved operators is much easier to do than enumerating all valid combination of reorderable operators. In many cases, it suffices to test syntactic conditions, when reordering operators is allowed. However, in the data model of XQuery, we often have to enumerate additional conditions that must hold in order to preserve the result order. As a very simple example, consider:

$$\chi_f(e_1 \times e_2) = \chi_f(e_1) \times e_2.$$

The first condition of Definition 2 holds because the map operator is linear and the cross product is linear in its left argument. Next, we notice that the equivalence is only correct if f only refers to attributes bound by e_1, i.e. $\mathcal{F}(f) \subset \mathcal{A}(e_1)$. Finally, we need to make sure that the result of function f in the subscript of the map operator does not depend on the position in the input sequence. Thus, we need to keep this information for each function to check the applicability of the equivalence.

Summarizing, we need to check both syntactic and semantic conditions when we want to reorder algebraic operators. Nevertheless, these conditions can usually be tested statically, i.e. without looking at any data.

In Figure 2.8, we summarize algebraic equivalences implied by the Definition. We focus on reordering the selection operator. But of course other linear operators can be reordered in the same fashion. The validity of these equivalences follows from the fact that linear operators are reordered and some simple conditions hold. For example, we need to check the syntactical correctness of the algebraic expressions on both sides of the equivalence, i.e. for Eqv. 2.6 we require that all attributes of p_1 are bound by expression e_1, i.e. $\mathcal{F}(p_1) \subset \mathcal{A}(e_1)$. In addition, we sometimes require that predicates or functions do not depend on the position in the input sequence. Hence, the validity check is much more general and extensible than enumerating all of the above equivalences. In Appendix A.1, we formally prove these equivalences and point out the necessity for checking semantic conditions.

We continue with algebraic equivalences that hold for algebraic operators which are not linear. Figure 2.9 contains several equivalences we prove here. The proofs of the equivalences discussed below can be found in Appendix A.1

Equivalence 2.13.

Preconditions None.

Basic Idea Since the union operator is not linear in any of its arguments, we need to be more careful when pushing predicates beneath the union operator. In particular, we need to push the selection into *both* arguments of the union operator.

Equivalence 2.14.

Preconditions $\mathcal{F}(p_2) \subset \mathcal{A}(e_2)$

Basic Idea While the Cross Product is not linear in its right argument, we can still push a selection into and out of its right argument. This important property is exploited by many algebraic query optimizers.

Equivalence 2.15.

Preconditions $\mathcal{F}(p_2) \subset \mathcal{A}(e_2)$

Basic Idea As a trivial but notable consequence of Eqv. 2.14, we can push selections into the right argument of the join.

Equivalence 2.16.

Preconditions $B \subset A$

Basic Idea We can remove adjacent projections and only keep the last, most restrictive one.

Equivalence 2.17.

Preconditions None.

Basic Idea After turning the unnest map operator into a d-join, we might subsequently be able to apply Eqv. 2.18.

Equivalence 2.18.

Preconditions e_1 and e_2 can be evaluated independently

Basic Idea After turning the d-join operator into a join, we can benefit from the associativity and linearity of the join. More efficient implementations are usually available for the join.

Binary Grouping

Some algebraic properties for binary grouping for algebras over sets were presented in [CM93, Ste95, ACJK01]. Here, we present algebraic equivalences that hold for our algebra over sequences of tuples.

Equivalence 2.19.

Preconditions $A_i \subset e_i$

Basic Idea For parallel processing, Equivalence 2.19 can be used to partition the grouping input, process the partitions independently, and union the result of grouping them. If we choose the partitions of the grouping input small enough to fit in main memory, we can evaluate the binary grouping operator for each partition in main-memory.

$$\sigma_p(e_1 \hat{\cup} e_2) = \sigma_p(e_1) \hat{\cup} \sigma_p(e_2) \quad (2.13)$$

$$\sigma_{p_2}(e_1 \times e_2) = e_1 \times \sigma_{p_2}(e_2) \quad (2.14)$$

$$\sigma_{p_2}(e_1 \bowtie_p e_2) = e_1 \bowtie_p \sigma_{p_2}(e_2) \quad (2.15)$$

$$\Pi_A(\Pi_B(e_1)) = \Pi_A(e_1) \quad (2.16)$$

$$\Upsilon_{\mathcal{A}(e_2):e_2}(e_1) = e_1 <e_2> \quad (2.17)$$

$$e_1 <e_2> = e_1 \times e_2 \quad (2.18)$$

$$e_1 \Gamma_{g;A_1\theta A_2;f} e_2 = \hat{\cup}_i(e_{1_i} \Gamma_{g;A_1\theta A_2;f} e_2) \quad (2.19)$$

$$(e_1 \Gamma_{g_1;A_1\theta_1 A_2;f_1} e_2) \Gamma_{g_2;A_1\theta_2 A_3;f_2} e_3 = (e_1 \Gamma_{g_2;A_1\theta_2 A_3;f_2} e_3) \Gamma_{g_1;A_1\theta_1 A_2;f_1} e_2 \quad (2.20)$$

$$e_1 \Gamma_{g;A_1=A_2;f} e_2 = \Pi_{\overline{A_2}}(e_1 \bowtie^{g:f(\epsilon)}_{A_1=A_2} (\Gamma_{g;=A_2;f}(e_2))) \quad (2.21)$$

Figure 2.9.: Algebraic equivalences for NAL

Equivalence 2.20.

Preconditions $\mathcal{F}(f_i) \subset \mathcal{A}(e_1) \cup \mathcal{A}(e_{i+1}), A_{1_i} \subset \mathcal{A}(e_1)$, and $A_j \subset \mathcal{A}(e_j), g_1 \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2), g_2 \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_3)$

Basic Idea This equivalence allows us to reorder the evaluation of adjacent binary grouping operators. It can be applied beneficially from left to right when the grouped values of e_2 consume a lot of memory. By reordering both grouping operations we can delay this resource consumption.

Equivalence 2.21.

Preconditions $A_i \subseteq \mathcal{A}(e_i)$, $A_1 \cap A_2 = \emptyset$, and $g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$.

Basic Idea As we will see in Chapter 4, binary grouping is applicable in a more general context. Since this operator is not widely used yet, it can be replaced by a sequence of unary grouping and outer join. Note however, that the predicate in the binary grouping operator is restricted to an equality predicate.

Outer Joins and Antijoins

Galindo-Legaria et al. [RGL90, GLR97] were the first to investigate algebraic equivalences that are valid or invalid for outer joins in an algebra over sets. Rao et al. [RLL+01] extended their results by integrating reorderability of joins, antijoins, and outer joins. In Figure 2.10 we repeat several of their results. Here, we restrict ourselfs to the left outer join. More equivalences exist when the right and full outer join is also included. Rao et al. [RLL+01] present a conflict matrix which summarizes the operators that cannot be reordered. From this matrix, we can derive invalid reorderings (see Figure 2.10). We also summarize the remaining valid equivalences including left outer joins and antijoins. Some of them require additional conditions to hold [GLR97].

We now discuss the valid equivalences for antijoins and outer joins.

Equivalence 2.27.

Preconditions $A_i \subseteq \mathcal{A}(e_i)$, A_i pairwise disjunct

Basic Idea A join operator can be pushed into or out of the left argument of an outer join.

Conflict Matrix:

Join	left argument	right argument
$\mathbb{N}$	$\{\triangleright(2.25)\}$	$\{\bowtie(2.22), \triangleright(2.23)\}$
$\triangleright$	$\{\mathbb{N}(2.25), \triangleright(2.26)\}$	$\{\mathbb{N}(2.23), \triangleright(2.26), \bowtie(2.24)\}$
$\bowtie$	$\{\mathbb{N}(2.22), \triangleright(2.24)\}$	$\{\}$

Invalid reorderings:

$$e_1 \mathbb{N}_{A_1\theta_1A_{2_1}} (e_2 \bowtie_{A_{2_2}\theta_2A_3} e_3) \neq (e_1 \mathbb{N}_{A_1\theta_1A_{2_1}} e_2) \bowtie_{A_{2_2}\theta_2A_3} e_3 \quad (2.22)$$

$$e_1 \mathbb{N}_{A_1\theta_1A_{2_1}} (e_2 \triangleright_{A_{2_2}\theta_2A_3} e_3) \neq (e_1 \mathbb{N}_{A_1\theta_1A_{2_1}} e_2) \triangleright_{A_{2_2}\theta_2A_3} e_3 \quad (2.23)$$

$$e_1 \triangleright_{A_1\theta_1A_{2_1}} (e_2 \bowtie_{A_{2_2}\theta_2A_3} e_3) \neq (e_1 \triangleright_{A_1\theta_1A_{2_1}} e_2) \bowtie_{A_{2_2}\theta_2A_3} e_3 \quad (2.24)$$

$$e_1 \triangleright_{A_1\theta_1A_{2_1}} (e_2 \mathbb{N}_{A_{2_2}\theta_2A_3} e_3) \neq (e_1 \triangleright_{A_1\theta_1A_{2_1}} e_2) \mathbb{N}_{A_{2_2}\theta_2A_3} e_3 \quad (2.25)$$

$$e_1 \triangleright_{A_1\theta_1A_{2_1}} (e_2 \triangleright_{A_{2_2}\theta_2A_3} e_3) \neq (e_1 \triangleright_{A_1\theta_1A_{2_1}} e_2) \triangleright_{A_{2_2}\theta_2A_3} e_3 \quad (2.26)$$

Valid reorderings:

$$(e_1 \bowtie_{A_1\theta_1A_{2_1}} e_2) \mathbb{N}_{A_{2_2}\theta_2A_3} e_3 = e_1 \bowtie_{A_1\theta_1A_{2_1}} (e_2 \mathbb{N}_{A_{2_2}\theta_2A_3} e_3) \quad (2.27)$$

$$(e_1 \mathbb{N}_{A_1\theta_1A_{2_1}} e_2) \mathbb{N}_{A_{2_2}\theta_2A_3} e_3 = e_1 \mathbb{N}_{A_1\theta_1A_{2_1}} (e_2 \mathbb{N}_{A_{2_2}\theta_2A_3} e_3) \quad (2.28)$$

$$(e_1 \bowtie_{A_1\theta_1A_{2_1}} e_2) \triangleright_{A_{2_2}\theta_2A_3} e_3 = e_1 \bowtie_{A_1\theta_1A_{2_1}} (e_2 \triangleright_{A_{2_2}\theta_2A_3} e_3) \quad (2.29)$$

$$e_1 \mathbb{N}_{A_1\theta A_2 \wedge p_1} e_2 = e_1 \mathbb{N}_{A_1\theta A_2} (\sigma_{p_1}(e_2)) \quad (2.30)$$

$$\sigma_{p_1}(e_1 \mathbb{N}_{A_1\theta A_2} e_2) = \sigma_{p_1}(e_1 \bowtie_{A_1\theta A_2} e_2) \quad (2.31)$$

Figure 2.10.: Reorderability of antijoin and outer join

Equivalence 2.28.

Preconditions $A_i \subseteq \mathcal{A}(e_i)$, A_i pairwise disjunct, $A_2\theta A_3$ rejects NULL values on e_2

Basic Idea Outer joins can be reordered, if their tuple preserving sides point into the same direction. Specifically for left outer joins, they can be reordered, if the right outer join rejects NULL values on its left argument. This property is also called Null-intolerant [RLL$^+$01] or strong.

Equivalence 2.29.

Preconditions $A_i \subseteq \mathcal{A}(e_i)$, A_i pairwise disjunct,

Basic Idea Joins can be pushed into and out of the left argument of an antijoin.

Equivalence 2.30

Preconditions $\mathcal{F}(p) \subseteq \mathcal{A}(e_2)$

Basic Idea A conjunct in the predicate of the outer join can be pushed into the outer join's right argument. Note that this is not the same as pushing a selection into the right argument. Note also that it is possible to push selections into the left argument of the outer join because the left outer join is linear in its left argument (see Eqv. 2.11).

Equivalence 2.31

Preconditions p_1 rejects NULL values on e_2

Basic Idea This is an important rewrite to simplify algebraic expressions containing outer joins. In [GLR97] this equivalence and a similar equivalence for two-sided outer joins is used to turn outer into single-sided outer joins or joins. This is beneficial because the join can be evaluated more efficiently, and more algebraic equivalences hold for the join than for the outer join.

2.2.4. Summary

In this section we have thoroughly investigated the algebraic properties of NAL, our logical algebra. Based on the notion of linearity, we have detected general rules that allow us to reorder operators in our algebra. Not surprisingly, it turned out that certain algebraic equivalences that hold for bags or sets are not valid any more for our algebra over sequences. Most importantly, cross products and joins are not commutative.

In our opinion, this severely constrains the search space considered by the plan generator. In Chapter 5, we discuss a number of experiments in which we have observed huge performance gains when we disregarded the order of sequences for join processing and establish the correct order after computing joins. Moreover, in many cases one can safely disregard the order of sequences, e.g. in arguments to aggregate functions. In these cases, optimizations known from the relational context can be applied. Hence, a complete derivation mechanism is needed that computes when order is not relevant any more [FHM+04, GRT07].

2.3. The Natix Physical Algebra

When we discuss optimizations on algebraic expressions of the logical algebra, we need to measure the effectiveness of alternative algebraic expressions. Thus, we need to associate algorithms with every operator in an algebraic expression. The set of all those algorithms that implement the algebraic operators in NAL represent the Natix Physical Algebra (NPA). We consider an optimization effective if the query evaluation plan (QEP) where our optimizations are applied evaluates faster than the one without our optimizations. For example, unnesting nested queries often increases the number of implementations that are available to the plan generator. Hence, it is important to supply the plan generator with efficient implementation alternatives for the operators in NAL to make unnesting really beneficial.

In this section we discuss all non-trivial implementations for operators in the NPA. For the remaining operators, we simply assume a direct mapping of the operator definition presented in Section 2.1 into code. The set of operators in the NPA is sufficient to efficiently evaluate every query we discuss in this thesis.

As a basis for our description, we need to arrange for some notation which helps us to decide which conditions must hold to apply an algorithm. Then we discuss the operator implementations in the same order as we did in Section 2.1 for the logical operators.

2.3.1. Architecture and Notation

Our physical algebra works on sequences of tuples. Each tuple contains a set of variable bindings representing the attributes of the tuple. Some physical operators extend these bindings by appending attribute-value pairs to a tuple.

In Natix we have extended the conventional iterator interface [Wes00]. In the conventional architecture [Gra93], the `open` and `close` routines perform initialization and deinitialization. To avoid this effort in some cases, we split each routine into three parts which do only parts of the work, e.g. allocation of memory, intialization of data structures, preparation of the first result tuple.

Tuples are passed from the argument operator to its consumer in a pipelined fashion. This means, we copy tuples only when it cannot be avoided. Many operators preserve the order of tuples they consume while producing their result. We only need to be careful when

$n \downarrow m$	m is child of n
$n \Downarrow m$	m is descendent of n
$n \uparrow m$	m is parent of n
$n \Uparrow m$	m is ancestor of n
$n \rightarrow m$	m is right sibling of n
$n \Rightarrow m$	m follows n
$n \leftarrow m$	m is left sibling of n
$n \Leftarrow m$	m precedes n
$n \cdot m$	m and n are the same node

Figure 2.11.: Notation for structural relationships

operators break the pipelined execution (pipeline breakers) and copy data into internal data structures, e.g. a hash table, and potentially return their input in a different order. In this section, we will investigate these issues in detail.

The set of supported operators we cover here comprises the common algorithms used to execute XQuery queries [BKHM05, PCS+05, RSF06].

Notation for Structural Relationships. To test structural relationships between XML nodes, we arrange for the notations in Figure 2.11. We also use these abbreviations to denote XPath axis steps. Then n is the context node, and m is a node reachable along the desired axis.

Properties of Predicates. To find the most efficient implementation for binary matching operators, we need to examine the properties of predicates. Therefore, we distinguish *symmetric, irreflexive* predicates ($\neq$) from *antisymmetric, transitive* predicates ($<, \leq, >, \geq$) and equivalence relations (i.e. *reflexive, symmetric, transitive*).

Properties of Aggregate Functions. Aggregate functions can be *decomposable* and *reversible* [CM95c]. These properties help us to find the most efficient implementation for binary grouping. We now recall their definitions.

The definitions are given in terms of sequences, but extensions to sets and bags are straightforward. Only the definitions of disjoint set union and the empty set need to be adjusted to the bulk type as follows:

bulk type	$\emptyset$	$\dot\cup$
set	empty set	disjoint set union
bag	empty bag	bag union
sequence	empty sequence	append sequence

Definition 3 (Decomposable Aggregate Function) *Let $\mathcal{N}$ be the codomain of a* scalar aggregate function $f : S \rightarrow \mathcal{N}$ *over some sequence S of tuples. We say $f : S \rightarrow \mathcal{N}$ is* decomposable *if there exist functions*

$$\begin{aligned} agg^I : S &\rightarrow \mathcal{N}' \\ agg^O : \mathcal{N}', \mathcal{N}' &\rightarrow \mathcal{N}' \\ agg^F : \mathcal{N}' &\rightarrow \mathcal{N} \end{aligned}$$

with

$$f(X) = agg^F(agg^O(agg^I(Y), agg^I(Z)))$$

for all non-empty sequences X, Y, and Z with $X = Y \mathbin{\dot\cup} Z$ and $Y \mathbin{\hat\cap} Z = \epsilon$.

$$
\begin{aligned}
f(X) &= agg^F(agg^O(agg^I(Y), agg^I(Z))) \\
count(X) &= id(fn : sum(fn : count(Y), fn : count(Z))) \\
sum(X) &= id(fn : sum(fn : sum(Y), fn : sum(Z))) \\
avg(X) &= div2(fn : sum(fn : sum(Y), fn : sum(Z)), \\
&\qquad fn : sum(fn : count(Y), fn : count(Z))) \\
min(X) &= id(min2(fn : min(Y), fn : min(Z)) \\
max(X) &= id(max2(fn : max(Y), fn : max(Z)))
\end{aligned}
$$

Figure 2.12.: Examples of decomposable aggregate functions

Decomposable aggregate functions allow us to aggregate on subsequences of the whole data and combine the results of these computations to the aggregate over the whole data.

Fortunately, many aggregate functions are decomposable. In Figure 2.12 we summarize how we can compute their value. In this figure, we use a prefix notation for addition or division to ease the transition from the abstract notation to the concrete aggregate function. The most difficult aggregate function is `fn:avg` where we have to both `fn:sum` and `fn:count` the values that belong to each sub sequence Y and Z. Both computations are combined by adding their results — this corresponds to function agg^O. To get the average for all values, we have to divide the sum and the count. For all aggregate functions but the average, we do not need function agg^F. In these cases we use the identity function `id` to denote that no function needs to be applied. We need to be careful with empty sequences. In these cases we use functions with suffix `2`. But we defer this special case until the end of this section.

We now define reversible aggregate functions that allow us to compute aggregates more efficiently.

Definition 4 (Reversible Aggregate Function) *A decomposable scalar function $f : S \to \mathcal{N}$ over some sequence S is called* reversible *if for agg^O there exists a function $(agg^O)^{-1} : \mathcal{N}', \mathcal{N}' \to \mathcal{N}'$ with*

$$f(Z) = agg^F((agg^O)^{-1}(agg^I(X), agg^I(Y)))$$

for all non-empty sequences X, Y, and Z with $X = Y \dot\cup Z$ and $Y \hat\cap Z = \emptyset$.

Reversible scalar aggregates allow us to compute the value of an aggregate function over some subsequence by computing the aggregate function over the sequence. Using this result, we can use the inverse function $(agg^O)^{-1}$ to compute the desired value for the subsequence.

In Figure 2.13 we show that `fn:sum`, `fn:count`, and `fn:avg` are reversible. The aggregate functions `fn:min` and `fn:max` are not reversible. To see this, consider

$$
\begin{aligned}
Y &= ([a : 1], [a : 2]) \\
Z &= ([a : 3], [a : 4]).
\end{aligned}
$$

We have $fn : min(Y) = 1$, $fn : min(X) = fn : min(Y \hat\cup Z) = 1$. Given these information, it is not possible to derive $fn : min(Z)$.

Empty Sequences. For some of these aggregate functions, we have to be careful with empty sequences. The functions `fn:min` and `fn:max` return an empty sequence when applied to an empty sequence. When we want to combine the result of computing the minimum or maximum of two sequences, we need to ignore this empty sequence. Hence, we define the function `max2` (and `min2` analogous) as:

$$
\begin{aligned}
f(Z) &= agg^F((agg^O)^{-1}(agg^I(X), agg^I(Y))) \\
count(Z) &= id(-(fn:count(X), fn:count(Y))) \\
sum(Z) &= id(-(fn:sum(X), fn:sum(Y))) \\
avg(Z) &= div2(-(fn:sum(X), fn:sum(Y)), -(fn:count(X), fn:count(Y)))
\end{aligned}
$$

Figure 2.13.: Examples of reversible aggregate functions

```
declare function max2($arg1 as xs:anyAtomicType?,
      $arg2 as xs:anyAtomicType?) as anyAtomicType?
{
  if (fn:empty($arg1))
  then $arg2
  else if (fn:empty($arg2))
       then $arg1
       else fn:max($arg1,$arg2)
};
```

The functions `fn:count` and `fn:sum` with one argument return the integer constant 0 when applied to empty sequences. Hence, we do not need special treatment there. Finally, function `fn:avg` returns the empty sequence for empty input. Thus, we use function `div2` to handle this case:

```
declare function div2($arg1 as xs:anyAtomicType,
      $arg2 as xs:anyAtomicType) as anyAtomicType?
{
  if ($arg2 eq 0.0)
  then ()
  else ($arg1 div $arg2)
};
```

All these problems are not unique to XQuery. For example in SQL, we have to take care of NULL values and different semantics of the function `COUNT`.

Partitioning. When we rely on decomposable or reversible aggregate functions we will usually partition the input by some grouping attribute. Thus, different groups might contain the same value to aggregate. However, for decomposable aggregate functions this does not cause any problems because all these aggregate functions are not sensitive to order. Hence, it does not matter in which order we combine the partial results. For decomposable aggregate function we note that $(agg^O)^{-1}$ must be the reverse function of agg^O. Since these functions exist for adding, subtracting, and counting but not for minimum or maximum the last two are not reversible.

2.3.2. Operator Implementations

Standard implementation techniques for some algebraic operators [Gra93] do not preserve order. In this section we survey the algorithms available in the NPA and comment on their properties concerning sequences.

Simple Operators

Several operators' definitions can be mapped trivially to an implementation. The first among them is the SINGLETONSCAN which returns a singleton sequence consisting of the empty tuple.

Similarly, the MAP operator $\chi_{a:e_2}(e_1)$ is implemented by evaluating its subscript e_2 for each tuple in e_1. The result of this evaluation is bound to variable a.

Sorting is a physical operator implementation for which there exists no counterpart in the logical algebra exists. We employ sorting to establish a user-defined order of the result or to establish the correct order after application of an operator that destroys the required order [MHKM04]. We support sorting in a list of attributes (denoted with SORT_a). Available sorting algorithms include Quicksort and Heapsort in main memory and external sorting based on replacement selection. We support stable sorting with these algorithms by introducing a tid which is appended at the end of the list of sort attributes.

Join Operators

Let us comment on the implementations of join operators. We do not discuss details of the join implementations. Instead, we summarize the preconditions that must hold to apply a join implementation and whether the join changes the order of its input. We refer to [DKO+84, ME92, Gra93, HCLS97] for surveys on implementing joins.

Figure 2.14 summarizes the join algorithms we used to implement the query evaluation plans. In this table we give the assumptions that must hold on the input of the join and on the predicates. The MERGEJOIN is the only algorithm that requires its input to be sorted on the join attributes. In most cases in XQuery this join cannot exploit document order of its input. Even worse, it requires sorting before *and* after its application. This severely limits the utility of this algorithm.

Several algorithms require the predicate to be an equivalence relation. In practice this usually means that the predicate must be an equality predicate. In particular the hash-based and sort-based algorithms require such a join predicate.

The right-hand side of the table shows to which extent order is preserved by our join implementations. In general, the nested-loop-based algorithms as well as the MERGEJOIN preserve the order of their left input. Furthermore, the predicate might induce functional dependencies and, hence, sorting of the right input might be preserved as minor order.

For both the HASHSEMIJOIN and HASHANTIJOIN, the order of their left input is preserved because this input is only filtered by a lookup in a hash table into which the right input is loaded. As the right input of either algorithm does not belong to the output, order-preservation of the right input is neither relevant nor meaningful.

The remaining hash-based join algorithms do not not preserve order of either input. Hence, we have no efficient join implementation available to implement equijoins. Fortunately, [CKK98] provide an efficient implementation for an order-preserving hash join. Further performance enhancements can be expected when using the order-preserving hash join. Until then, we employ nested-loop-based algorithms or resort to the techniques described in [MHKM04] and repair order.

Set Operators

Union as defined in Section 2.1 can is implemented as simple concatenation of two sequences of tuples. Of course, we have to remove duplicates from both inputs when we compute the union of two sequences of nodes in an XPath expression. In this case, a merge-based union operator should be used that merges its two arguments based on their input order. Currently, we have not implemented this operator but use a sort-based duplicate elimination operator after the sorting the unioned sequences by document order.

We implement union and difference as defined in Section 2.1 in terms of semijoin and antijoin. Fortunately, we do note take care of duplicates for those operations on node sequences when the first sequence is in document order and duplicate free.

Algorithm Name	Assumptions			Preserves Order	
	e_1	e_2	$A_1\theta A_2$	e_1	e_2
DEPENDENTNLJOIN	-	-	-	$\surd$	($\surd$)
NLJOIN	-	-	-	$\surd$	($\surd$)
SIMPLEHASHJOIN	-	-	E	-	-
GRACEHASHJOIN	-	-	E	-	-
MERGEJOIN	S	S	E	$\surd$	($\surd$)
θ-SEMIJOIN	-	-	-	-	-
θ-ANTIJOIN	-	-	-	-	-
HASHSEMIJOIN	-	-	E	$\surd$	-
HASHANTIJOIN	-	-	E	$\surd$	-
NLLEFTOUTERJOIN	-	-	-	$\surd$	($\surd$)
HASHLEFTOUTERJOIN	-	-	E	-	-

S sorted **E** equivalence relation

Figure 2.14.: Implementations for join operators $e_1 \bowtie_{A_1\theta A_2} e_2$

Grouping and Duplicate Elimination Operators

Unary Grouping and Duplicate Elimination. We support both a hash-based and a sort-based version of the unary grouping operator. Duplicate elimination is implemented as a special case of either algorithm. The hash-based grouping operator aggregates the values of each group in a main-memory hash table and does not preserve insertion order. The implementation of the sort-based grouping operator is straight forward [Gra93]. For grouping or duplicate elimination on very large data sets, we prefer the sort-based implementations. In future, we may also support materializing hash-based implementations as described in [BD83, HNM02].

Binary Grouping. In Chapter 4 we will examine the value of the binary grouping operator for unnesting nested queries. Here, we survey implementations of this operator which can efficiently evaluate the algebraic pattern $\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1)$ [ACJK01, CM95c, MM05a, MM05b].

In the context of XQuery, we have investigated the implementations summarized in Figure 2.15 (see [MM05b] for details). The left part of the table contains the algorithms with their simplified time and space complexity. In these complexity formulas we denote with $n = max(|e_1|, |e_2|)$.

The right part of the table surveys the assumptions for each algorithm. Thus, it can be used as a guide to the most efficient implementation. The assumptions are related to the inputs e_1 and e_2, the predicate $A_1\theta A_2$, and the function f in the binary grouping operator $e_1\Gamma_{g;A_1\theta A_2;f}e_2$ defined in Section 2.1. The last column indicates the ratio of improvement in execution time over the direct nested evaluation of the nested query using NESTEDSORT. For some algorithms ranges for Δ are given. Values of $\Delta > 1.0$ indicate an improvement by a factor Δ. Obviously, algorithms with more assumptions evaluate up to three orders of magnitude faster than the nested-loops-based algorithms with fewer assumptions. The algorithms at the bottom of the table perform in linear time, compared to quadratic time in the general case of nested evaluation. In general, algorithms that require sorted input demand constant space, while hash-based algorithms use linear space in the size of the grouping input.

To be able to evaluate XQuery queries, we have extended these implementations to support sequences [MM05b]. Basically, we keep duplicates when detecting groups and use a helper data structure to record the order of insertion into the hash table.

Algorithm			Assumptions				
Name	Time	Space	e_1	e_2	$A_1\theta A_2$	f	Δ
NESTED	$O(n^2)$	$O(n)$	-	-	-	-	0.95-1.2
NLBINGROUP	$O(n^2)$	$O(n)$	-	-	-	-	0.65-0.75
HASHBINGROUP	$O(n \lg n)$	$O(n)$	-	-	¬SY, T	D	1300
TREEBINGROUP	$O(n \lg n)$	$O(n)$	-	-	¬SY, T	D	1300
EQBINGROUP	$O(n)$	$O(n)$	-	-	SY	RE	1850
NESTEDSORT	$O(n^2)$	$O(1)$	S	-	-	-	1.0
SORTBINGROUP	$O(n^2)$	$O(1)$	S	-	-	-	1.1-1.2
LTSORTBINGROUP	$O(n)$	$O(1)$	S	S	¬SY, T	-	2100

S sorted **SY** symmetric **D** decomposable

T transitive **RE** reversible

Figure 2.15.: Assumptions and complexity for the implementations of the binary grouping operator

XPath Evaluation Operators

Navigating Implementations. The definition of the unnest map operator in Section 2.1 suggests that the result of evaluating the subscript is temporarily materialized before it is returned as a result. Our implementation of the UNNESTMAP is more efficient because it evaluates the subscript in a lazy fashion and immediately returns a result tuple computed in the subscript. The UNNESTMAP operator is mostly used to evaluate XPath location steps. Thus, the subscript e_2 contains the location step $c/a :: n$ resulting in $\Upsilon_{cn:c/a::n}(e_1)$. Given a context node stored in variable c of a tuple $t \in e_1$, it evaluates axis a and applies node test n to the remaining candidate nodes. Each result node is bound to variable cn in the result tuple. During the evaluation of a location step the operator navigates through the document that potentially contains result nodes. This traversal is done for every context node. Note that the result nodes for each context node are generated in document order.

Index-Aware Operators. For efficient XPath evaluation [SAKJ+02] proposed the STRUCTURALJOIN ($e_1 \bowtie_p^{ST-J} e_2$). It joins one sequence of tuples of context nodes, e_1, with a sequence of candidate nodes, e_2. Both sequences must be sorted in document order. Predicate p tests the axis step relation that must hold between nodes of the two sequences as summarized in Figure 2.11. We perform these tests on ORDPATH IDs [OOP+04] which we use as logical node identifiers. We do not use any implementation of the Holistic Twig Join [BKS02] because we optimize XPath expressions on a fine-grained level.

Index Operators

We employ the INDEXSCAN ($Idx_{n;A;p;rp}$) to access data stored in a B-link tree named n. A is a set of attribute bindings established by the scan. It must be a subset of the attributes defined in the schema of the index. Predicate p optionally tests the upper and lower bound for the range scan over all leaf pages of the index. The residual predicate rp is an optional predicate applied to each tuple before it is passed to the consumer operator. We use the INDEXSCAN to retrieve logical XML node identifiers to be used by the STRUCTURALJOIN.

The index we use in Natix is an implementation of a B^+-tree with sibling links based on the algorithms proposed by Jaluta et. al [JSSS05, BÖ5]. This B-link index allows storing keys of variable size and performs online rebalancing and deallocation operations in case of underutilized nodes. These operations are especially beneficial as the index performance does not deteriorate due to document updates, and explicit garbage collection operations

become obsolete. We employ this index to support the evaluation of location steps in path expressions. Thus, another useful feature for processing ORDPATH IDs includes key-range scans exploiting the sibling links and high concurrency by restricting the number of latches to a minimum.

2.4. Related Work

Logical Algebra

Unfortunately, there is no standard logical algebra for XQuery or XPath yet as it is available to query relational databases. So far, three main camps have emerged:

Extensions to Relational Algebras. The first camp leverages the power and experiences of optimizing OQL and SQL by extending the logical algebras used for these languages. Relational algebras are based on sets [Mai83]. For SQL, this algebra was extended to support bag semantics [DGK82, Alb91]. Further extensions were needed to support OQL [CM93, SABdB94], e.g. relation-valued attributes [RKS88]. Because the XQuery data model is based on sequences of items, algebras for XQuery need to handle both duplicates and order. Algebras proposed for order- and duplicate-aware data models [SJS02, LS03] and specifically for XQuery include [BT99, FHP02, VGD$^+$02, MHM03a, GT04]. They all have in common that (1) they extend the relational algebra with new operators needed to translate XQuery and (2) treat order and duplicates of the data.

Among the proposed algebras only [SJS02, FHP02, VGD$^+$02, LS03] list algebraic equivalences valid for their algebras. However, all of them omit rigorous correctness proofs. To the best of our knowledge, this is the first extensive treatment of algebras over sequences.

Tree Algebras. The second camp represents queries as pattern trees [JLST02]. XPath expressions are translated into a pattern tree. Computing the result of a XPath expression corresponds to finding all embeddings of the tree pattern in the XML tree instance.

The use of tree algebras is motivated by the fact that one can formally reason about trees [Suc01]. Theoretical results on tree patterns include e.g. satisfiability of path queries [Hid03], query containment [MS02, Sch04], or tree pattern minimization [ZO02, AYCLS01]. A particularly interesting result is that query containment is $co\mathcal{NP}$ complete once either two features $//, [], *$ are combined with the child axis [MS02]. For more restricted cases query containment is in $\mathcal{P}$. As query containment is an important test for applicability of views to answer a query, these results affect our translation procedure discussed in Chapter 3.

On the other hand, tree algebras seem to lack the expressiveness needed to represent any query formulated in XQuery. Most tree algebras are restricted to a subset of axis steps and have difficulty in expressing advanced XQuery constructs such as node construction or type-based constructs.

Calculus Representations. The third camp translates the XQuery query into a representation close to the query language level. This includes representations as query graph [JK84, HFLP89] or in comprehension calculus [FM95, FM00]. Both the query graph model [SKS$^+$01, FKS$^+$02, OCP$^+$05] and the comprehension calculus [FLBC02] required extensions to support XQuery. The advantage of a calculus representation is that it is more declarative than an algebra expression. As a consequence, optimizations such as unnesting rewrites are easier to implement because pattern matching needs to consider fewer cases. On the other hand, another translation step is needed because a query evaluation plan is usually expressed as an algebraic expression.

Other Approaches. In the literature on XQuery optimization the distinction between logical and physical algebra is often blurred [BKHM05, RSF06]. The reason is that heuristics are used to directly derive an efficient QEP from the query. In this work we prefer to clearly separate logical and physical algebra, as it is done e.g. in [JAKC+02, FHK+04, LKA05, BGvK+06]. While several optimizations are valid and useful on the logical algebra, e.g. simplification of algebraic expressions or normalization, our final goal is to find a cost-efficient query evaluation plan based on costs. This requires to associate costs to each primitive of QEP and to compute cardinalities for the input to each algebraic operator. Up to now this information is only available for a subset of these primitives.

Physical Algebra

The architecture of our physical algebra is based on iterators [Gra93, Wes00]. Operators treat their input as a sequence of tuples and, hence, preserve the order of their input tuples in many cases. However, some operators alter the order of their input. In this section we have examined this problem for joins, grouping, and duplicate elimination.

A large number of techniques for efficient XPath evaluation were proposed, e.g. [ILW03, LMP02, SAKJ+02, BKS02, Gru02, GKP02, GKP05, JFB05, KBM05]. Several proposals for XPath evaluation require specialized index structures, e.g. [GW97, LM01, CVZ+02, BCM05]. If not explicitly mentioned otherwise we either employ the canonical or the stacked translation (and evaluation) of XPath expressions presented in [BKHM05]. We have chosen this rather simple evaluation technique because (1) it does not require any additional indices, (2) it can be used to evaluate any kind of XPath expression, and (3) there is no satisfying cost model yet to choose any of the other techniques. Our analysis shows that this evaluation technique is superior to most other techniques when large parts of the document must be visited to answer a query [MBB+06].

3. Translating XQuery into the Algebra

The equivalences discussed in Chapter 2 transform algebraic expressions. As we have motivated in this chapter, algebraic equivalences provide us with a formal framework to reason about the correctness of transformations on algebraic expressions. At the same time, the algorithms that implement the operators in NAL lead us to different costs for query evaluation strategies enumerated in a cost-based optimizer.

To be able to benefit from this algebraic framework, we need to translate XQuery statements into our algebra. As we will see in Section 3.4, we do not directly translate XQuery into NAL, but apply a normalization step first. This simplifies our translation procedures which we present in Section 3.5. While the result of the translation procedure presented here will be an algebraic expression, the actual implementation consists of a mixture of algebra and calculus. In Section 3.6, we describe the mapping of our NAL algebra into this representation.

All steps presented in this chapter are implemented in the NFST module of our query optimizer. Evidently, we concentrate on **N**ormalization and **T**ranslation. We discuss **F**actorization of common subexpressions only shortly in Section 3.6 where we present our internal representation and **S**emantic Analysis in Section 3.7 in the context of typing.

3.1. Relevant XQuery Fragment

In this work, we focus on a subset of XQuery that is expressive enough to formulate complex queries, e.g. nested queries. However, the translation and optimization approach we cover here is general enough to support the missing features. Figure 3.1 presents the subset of the XQuery grammar we currently support. It is a variant of LixQuery grammar [HPVD04].

In this grammar, we denote terminals with `terminal` and non-terminals with $\mathrm{nonterminal}$. Some terminals contain complex regular expressions of tokens. We refer to [HPVD04] for their definition and simply use angle brackets instead, i.e. $<\mathrm{complex\ token}>$. For simplicity, we use a very restrictive set of functions which we all treat as special built-in functions. In particular, we ignore user-defined or recursive functions.

In the grammar, we only give the productions for computed constructors. Since our example queries use direct constructors, we need to normalize them into computed constructors as defined in [DFF$^+$07]. We will use this normalization step in this thesis without repeating the associated rewrites.

We have added `flwrExpr` to be able to express queries more succinctly and `quantExpr` because we want to express quantifiers explicitly. Furthermore, we distinguish between general comparison – having existential semantics – and value comparison. All these extensions to LixQuery are syntactic sugar, but are often used in practice. As we will see later, their treatment has several implications on normalization, translation, and optimization of XQuery.

Note that we have simplified the grammar. For example, our grammar does not explicitly enforce any precedence rules for binary operators as it is done in the XQuery specification [DFF$^+$07]. Nevertheless they are still left associative.

```
mainModule   ::=  expr <EOF>
expr         ::=  singleExpr | exprSeq
exprSeq      ::=  singleExpr ( "," singleExpr )*
singleExpr   ::=  flwrExpr | quantExpr | andExpr
builtIn      ::=  ( "doc(" singleExpr ")"
                  | "name(" singleExpr ")"
                  | "string(" singleExpr ")"
                  | "integer(" singleExpr ")"
                  | "contains(" singleExpr "," singleExpr ")"
                  | "true()" | "false()"
                  | "not(" singleExpr ")"
                  | "count(" singleExpr ")"
                  | "distinct-values(" singleExpr ")"
flwrExpr     ::=  (forExpr | letExpr)+ whereClause? "return" singleExpr
rangeExpr    ::=  var "in" singleExpr
bindExpr     ::=  var ":=" singleExpr
forExpr      ::=  "for" rangeExpr ("," rangeExpr)*
letExpr      ::=  "let" bindExpr ("," bindExpr)*
whereClause  ::=  "where" singleExpr
quantExpr    ::=  ("some" | "every") rangeExpr ("," rangeExpr)*
                  "satisfies" ExprSingle
andExpr      ::=  compExpr ( ("or" | "and") compExpr )?
compExpr     ::=  addExpr ( (genComp | valComp | nodeComp) addExpr )?
genComp      ::=  "=" | "!=" | "<" | "<=" | ">" | ">="
valComp      ::=  "eq" | "ne" | "lt" | "le" | "gt" | "ge"
nodeComp     ::=  "<<" | ">>" | "is"
addExpr      ::=  multExpr ( ("+" | "-") multExpr )*
multExpr     ::=  union ( ("*" | "div" | "idiv" | "mod") union )*
union        ::=  path ( ("|" | "union" | "intersect" | "except" ) union )*
path         ::=  filter ( ( "/" | "//" ) path )*
filter       ::=  step ( "[" singleExpr "]" )*
step         ::=  "." | ".." | qname | "@" qname | "*" | "@*"
                  | "text()" | primaryExpr
primaryExpr  ::=  builtIn | qname | constr | var | literal | empSeq | "(" expr ")"
literal      ::=  string | integer
string       ::=  <String>
integer      ::=  ( (<Digits> | "+" <Digits> ) | ("-" <Digits> ) )
var          ::=  "$" qname
empSeq       ::=  "()"
constr       ::=  ( "element" "{" singleExpr "}" "{" expr "}"
                  | "attribute" "{" singleExpr "}" "{" expr "}"
                  | "text" "{" singleExpr "}"
                  | "document" "{" singleExpr "}"
qname        ::=  <NCName> | (<NCName> ":" <NCName>)
```

Figure 3.1.: BNF of the supported XQuery fragment

3.2. Requirements

The operations discussed in this chapter are important as a preparation for the subsequent phases of the algebraic optimizer. Thus, we require the following properties of normalization and translation:

Soundness Each transformation must preserve the semantics of the given query.

Completeness Ideally, every query construct should be handled by the normalization and translation step. As XQuery is a query language with many features, we cannot treat every language construct in this work. Instead, we concentrate on the XQuery fragment defined in Section 3.1.

Uniqueness The normalization and translation of equivalent queries should result in the same representation of the query, i.e. a normal form. The application order of rewrite rules should not matter, and it should be ensured that the normal form is reached by the normalization algorithm if the normal form exists.

Effectiveness The normal form should be reached in a finite number of transformations, preferably in a few transformation steps. We will point out how we achieve this.

Optimizability The normal form should be a good starting point for later optimization steps. We will see in Chapter 4 that our translation procedure provides a good foundation for optimization.

3.3. Notation

Conceptually, the normalization and translation rules we present here match patterns of the textual XQuery representation and transform them, given the bindings of the matched pattern. In this section, we will denote pattern matching with regular expressions on the grammar presented in Section 3.1. We refer to terminal symbols with `terminal` and to non-terminals with $\mathrm{nonterminal}$. During normalization, some rules introduce new variable names using the expression $< \$v = newVar() >$. Thereby, we create a new variable name to which we can refer by $\$v$. We will also use fun to refer to arbitrary functions including $\mathrm{builtIn}$, $\mathrm{andExpr}$, $\mathrm{compExpr}$, $\mathrm{addExpr}$, $\mathrm{multExpr}$, and constr. In the case of constructors, these arguments refer to the computed node name and the computed content.

3.4. Normalization

As an important preparation step to the translation, we normalize XQuery expressions on the query level. More precisely, all normalization steps work on the abstract syntax tree created by the XQuery parser. Our normalization rewrites transform the XQuery statement into a normal form. The normalized query is easier to translate the query into our algebra because we have to consider fewer query patterns. Moreover, normalization facilitates common subexpression elimination because we introduce new variables that are bound to complex expressions. In this query representation, it is much easier to detect common subexpressions.

There are many relationships between path expressions embedded into XQuery expressions and equivalent expressions in XQuery [DFF$^+$07, JHSV06]. For example, the transformation of XQuery into the XQuery core breaks location steps into nested FLWOR expressions [DFF$^+$07]. We use several of these techniques and, hence, reuse normalization rules presented there. But we will tailor normalization for our needs. In particular, we will break XPath expressions apart only when a location step contains a filter expression. The reverse step, detecting tree patterns, has been discussed in [JHSV06]. Our motivation

for doing so is that especially for simple path expressions many optimizations are known, e.g. [AYCLS01, ZO02, HKM02, HM03, BOB+04]. Several of these optimizations are only tractable or applicable for simple path expressions.

First, we discuss our normalization steps for FLWR expressions. Afterwards, we treat XPath expressions.

3.4.1. FLWR Expressions

Objectives

The objective of normalizing FLWR expressions consists in obtaining a uniform representation for different formulations of the FLWR expression. As a result, the subsequent steps of query compilation are simplified, most importantly the translation step and several optimizations. For example, during normalization we reduce the number of query patterns which have to be handled during query translation. Our normalization rewrites separate the query into three parts:

The binding part consists of **for** and the **let** clauses. It gathers all queried data, computes intermediate results, and binds them to variables.

The modifying part alters the tuple stream, either by changing the order of items as specified in the **order by** clause or by filtering out items in the **where** clause.

The result construction part consists of the the **return** clause, which solely refers to bound variables.

Normalization Rules

In Figure 3.2, the rewriting rules for normalizing FLWR expressions are summarized. We now discuss the idea behind each normalization rule.

N-3.1 and N-3.2 We split **for** or **let** clauses that bind multiple variables into individual clauses. Note that the occurrence indicator of the clause in both rules is $+$ instead of $*$, as in the grammar productions for the $\mathrm{forExpr}$ and $\mathrm{letExpr}$. This is necessary for the correctness of the rewrite because it makes sure that the list of **for** or **let** clauses contains at least two clauses. After the exhaustive application of this rewrite, each $\mathrm{forExpr}$ or $\mathrm{letExpr}$ binds at most one variable.

N-3.3 We split quantified expressions that bind multiple variables into individual quantified expressions. Note that the occurrence indicator on the $\mathrm{RangeExpr}$ in both rules is $+$ instead of $*$, as in the grammar production for the $\mathrm{quantExpr}$. As for the previous rewrites it is necessary for the correctness of the rewrite. After the exhaustive application of this rewrite, each $\mathrm{quantExpr}$ contains at most one $\mathrm{RangeExpr}$.

N-3.4 When the **where** clause of a FLWR expression contains a complex expression, we introduce a new $\mathrm{letExpr}$ and bind the computation of this complex expression to a new variable \$p. For this rewrite, we consider $\mathrm{singleExpr}_1 \in \{\mathrm{flwrExpr}, \mathrm{builtIn}, (\mathrm{Expr})\}$ as complex expressions but leave comparisons and quantified expressions as they are. We replace the old complex expression by a reference to the new variable \$p. After the exhaustive application of this rewrite, the **where** clause contains only references to variables, quantified expressions, or comparison operators.

N-3.5 We move every complex expression in the range predicate of a quantified expression into a new $\mathrm{letExpr}$. These $\mathrm{letExpr}$ are a convenient extension to simplify detection of common subexpressions and during translation. For this rewrite, we consider $\mathrm{singleExpr} \in \{\mathrm{flwrExpr}, \mathrm{builtIn}, (\mathrm{Expr})\}$ as complex expressions.

N-3.6 Similar to rule N-3.4, we move a complex expression from the **return** clause of a FLWR expression into a new letExpr. For this rewrite, we consider singleExpr $\in$ {flwrExpr, builtIn, (Expr), constr, exprSeq} as complex expressions. The exhaustive application of this rewrite leaves only a single variable reference in the **return** clause.

N-3.7 This rule replaces complex expressions inside a sequence of expressions by variables which are bound to the result of the replaced complex expression. We consider singleExpr $\in$ {flwrExpr, builtIn, (Expr), constr, exprSeq} as complex expressions.

N3.8 This rule replaces complex expressions as function arguments by variable references which are bound to the result of the replaced complex expression. We consider singleExpr $\in$ {flwrExpr, builtIn, (Expr), constr, exprSeq} as complex expressions. We also treat built-in functions, constructors, arithmetic expressions, or comparisons as functions and refer to them by fun.

N-3.9 This rule turns general comparisons denoted by genComp into value comparisons denoted by valComp.

The original general comparison is replaced by a quantified expression with the corresponding value comparison. The mapping of general comparisons into value comparisons is summarized in the table below (see also [DFF+07]). Note that we introduce the proper type conversion while typing both arguments and, hence, do not introduce them here.

genComp	valComp
=	eq
!=	ne
<	lt
<=	le
>	gt
>=	ge

Let us make sure that the rules in Figure 3.2 achieve our goals. First, notice that neither in the **where** clause nor in the **return** clause any of the rewrites introduces complex expressions. Second, notice that the rewrites introduce complex expressions only in new **let** clauses. They possibly create new **for** or **let** clauses containing complex expressions.

The exhaustive application of these rewrites eventually results in the normal form discussed at the beginning of this section. Since we only move around complex expressions but do not create ones, we reach this normal form in as many steps as there are complex expressions.

3.4.2. XPath Expressions

When normalizing XPath expressions, our main goal consists in restructuring them such that they are easier to optimize. We attempt this by breaking branching path expressions into simple path expressions. This gives us two important opportunities for optimization: (1) Predicates become visible. This enables us to detect join predicates, to move them into the **where** clause, and to unnest nested XPath expressions. (2) We assume that indices or materialized views are available rather for simple path expressions than for complex path expressions. Additionally, the problem of matching view definitions to path expressions in the user query becomes tractable when we extract simple path expressions from complex path expressions.

However, we have to be careful to preserve the semantics of path expressions.

1. In particular, we need to preserve document order, and we need to handle duplicates and position-based functions correctly. Currently, we do not rewrite the XPath expression when its evaluation depends on document order.

2. XPath expressions can contain predicates that correlate the selected node in the current path expression to nodes in another path expression.

$$\texttt{for}\ \text{rangeExpr}_1\ (,\ \text{rangeExpr})+ \rightarrow \texttt{for}\ \text{rangeExpr}_1\ \texttt{for}\ \text{rangeExpr}(,\ \text{rangeExpr})* \quad (3.1)$$

$$\texttt{let}\ \text{bindExpr}_1\ (,\ \text{bindExpr})+ \rightarrow \texttt{let}\ \text{bindExpr}_1\ \texttt{let}\ \text{bindExpr}(,\ \text{bindExpr})* \quad (3.2)$$

$$(\texttt{some}|\texttt{every})\ \text{rangeExpr}_1\ (,\ \text{rangeExpr})+\ \texttt{satisfies}\ \text{exprSingle} \rightarrow (\texttt{some}|\texttt{every})\ \text{rangeExpr}_1\ \texttt{satisfies}\ (\texttt{some}|\texttt{every})\ \text{rangeExpr}\ (,\ \text{rangeExpr})*\ \texttt{satisfies}\ \text{exprSingle} \quad (3.3)$$

$$(\text{forExpr}|\text{letExpr})+\ \texttt{where}\ \text{singleExpr}_1\ \texttt{return}\ \text{singleExpr}_2 \rightarrow (\text{forExpr}|\text{letExpr})+\ \texttt{let} < \$p = newVar() > := \text{singleExpr}_1\ \texttt{where}\ \$p\ \texttt{return}\ \text{singleExpr}_2 \quad (3.4)$$

$$(\texttt{some}|\texttt{every})\ \text{rangeExpr}\ (,\ \text{rangeExpr})*\ \texttt{satisfies}\ \text{singleExpr} \rightarrow (\texttt{some}|\texttt{every})\ \text{rangeExpr}\ (,\ \text{rangeExpr})*\ \texttt{let} < \$v = newVar() > := \text{singleExpr}\ \texttt{satisfies}\ \$v \quad (3.5)$$

$$(\text{forExpr}|\text{letExpr})+\ \text{whereClause?}\ \texttt{return}\ \text{singleExpr} \rightarrow (\text{forExpr}|\text{letExpr})+\ \texttt{let} < \$v = newVar() > := \text{singleExpr}\ \text{whereClause?}\ \texttt{return}\ \$v \quad (3.6)$$

$$\texttt{let}\ \text{var} := (\text{singleExpr}_1(,\ \text{singleExpr})+) \rightarrow \texttt{let} < \$v = newVar() > := \text{singleExpr}_1\ \texttt{let}\ \text{var} := (\$v(,\ \text{singleExpr})+) \quad (3.7)$$

$$\texttt{let}\ \text{var} := fun(\text{expr}) \rightarrow \texttt{let} < \$v = newVar() > := \text{expr}\ \texttt{let}\ \text{var} := fun(\$v) \quad (3.8)$$

$$\text{singleExpr}_1\ \text{genComp}\ \text{singleExpr}_2 \rightarrow \texttt{some} < \$v_1 = newVar() > \texttt{in}\ \text{singleExpr}_1\ \texttt{let} < \$v_2 = newVar() > := \texttt{data}(\$v_1)\ \texttt{satisfies}\ \texttt{some} < \$v_3 = newVar() > \texttt{in}\ \text{singleExpr}_2\ \texttt{let} < \$v_4 = newVar() > := \texttt{data}(\$v_3)\ \texttt{satisfies}\ \$v_2\ \text{valComp}\ \$v_4 \quad (3.9)$$

Figure 3.2.: Normalization of FLWR expressions

$$\text{path}_{1_{\$c}}(/|//)\ \text{step}\ [\text{singleExpr}] \rightarrow \texttt{for} < \$v = newVar() > \texttt{in}\ \$c\ /\ \text{path}\ (/|//)\ \text{step}\ \texttt{where}\ \text{singleExpr}\ \texttt{return}\ \$v \quad (3.10)$$

$$\text{path}_{1_{\$c}}\ (/|//)\ \text{step}\ ([\text{singleExpr}])+\ \text{path}_2 \rightarrow \texttt{let} < \$v_1 = newVar() > := \$c\ /\ \text{path}_1\ (/|//)\ \text{step}\ ([\text{singleExpr}])+\ \texttt{for} < \$v_2 = newVar() > \texttt{in}\ \$v_1/\ \text{path}_2\ \texttt{return}\ \$v_2 \quad (3.11)$$

Figure 3.3.: Normalization of XPath expressions

3. XPath expressions may contain nested expressions that are interpreted as nested queries.

Our normalization rewrites are summarized in Figure 3.3. They introduce new variables that store the intermediate results of the path expressions. We expect this to be beneficial for factorization of common subexpressions. When we add these variables into the current scope, we have to avoid name clashes.

N-3.10 This rewrite moves an XPath predicate into the **where** clause of a FLWR expression when the path expression is inside a **for** clause.

Note that we ignore several intricate issues here: (1) the result of the XPath predicate is the effective boolean value of expression exprSingle. The computation done for

the predicate might depend on actual types returned at runtime. (2) Positional predicates are another source of difficulty we ignore here. (3) Document order must be correct, e.g. when the last axis step before a positional predicate computes a reverse axis.

N-3.11 This rewrite allows us to break XPath expressions into pieces.

Note that in both rewrites we use the variable $\$c$ to explicitly refer to the set of context nodes. We also expand abbreviated syntax in path expressions into the corresponding unabbreviated form [DFF+07], i.e.

1. We treat occurrences of `@NodeTest` as attribute axis, i.e. `attribute::NodeTest`.
2. We treat occurrences of `..` as parent axis, i.e. `parent::node()`.
3. We expand each occurrence of `//` in a relative location path to `/descendant-or-self::node()/`. When the axis step afterwards contains a node test but no positional predicate, we can even replace `//NameTest` by `/descendant::NameTest`, which is more efficient to evaluate.
4. We rewrite absolute location paths so that they explicitly use function `fn::root`, i.e. `fn:root(self::node()) treat as document-node()/`.
5. When the axis name is omitted from an axis step, the default axis is child unless the axis step contains an AttributeTest or SchemaAttributeTest. Hence, we expand these path expressions by a child step including the node test.

3.4.3. Example Queries

In this section, we apply our normalization rules to concrete queries to demonstrate their effectiveness in establishing our normal form. We focus on queries containing quantified expressions or implicit grouping for two reasons. First, we get existentially quantified expressions when we normalize general comparisons when we make implicit computations explicit. Second, we want to rewrite to prepare the queries such that they are more convenient to optimize. In particular, both types quantified expressions and implicit grouping is formulated with nested queries. In Chapter 4, we show how to unnest this important class of queries.

General Comparisons

First, we look at a query that contains a general comparison. This is an interesting example because the general comparison implicitly has existential semantics.

```
for $t1 in doc("bib.xml")//book/ title
where $t1 = doc("reviews.xml")// entry / title
return $t1
```

Normalization is simple because we only need to turn the general comparison into a quantified expression using rewrite N-3.9. This rewrite introduces function `data` to apply atomization to the result of both range expressions.

$\xrightarrow{N-3.9}$

```
for $t1 in doc("bib.xml")//book/ title
where some $v1 in $t1
        let $v2 := data($t1)
        satisfies
       some $v3 in doc("reviews.xml")// entry / title
```

```
      let $v4 := data($v3)
      satisfies $v2 eq $v4
return $t1
```

We now have established the desired form:

1. All data retrieval is done in the **for** or **let** clauses.
2. Implicit computations (e.g. the existential nature of general comparison) have become explicit.
3. The **return** clause only contains variable references.
4. Function calls, except function `fn:distinct-values`, do not contain complex expressions as arguments.

Implicit Grouping

Since there is no explicit grouping construct in XQuery yet, grouping must be formulated with nested queries. Because grouping in XQuery can be expressed in many ways, we like to treat all of them in a uniform fashion. Among the different possibilities to express grouping implicitly, one can use a nested FLWR expression in the **return** clause. Alternatively, it is possible to use a nested FLWR expression in the **let** clause. Queries containing grouping often employ element constructors to mimic tuples [BCC+04, BC04]. The reason is that the XQuery data model is based on *flat* sequences of items, i.e. sequences cannot be nested.

For these reasons, we investigate the query below which features all these concepts. Normalization of this query starts by removing the element constructor from the **return** clause (rewrite N-3.6). Remember that it is our goal to have only variable references in the **return** clause. Next, we replace the sequence of expressions in the element constructor by a variable reference and introduce a new **let** clause. Therefore, we apply rewrite N-3.8. Note that we treat the constructor as a function here. Then, we replace the two complex expressions inside the sequence expression by variables using rewrite N-3.7.

Finally, we normalize the FLWR expression bound to the new variable $\$v4$. As before, we introduce a new **let** which is bound to the path expression in the **return** clause (rewrite N-3.6). Then, we move the XPath predicate into the **where** clause. In the next normalization step, we extract the path expression in the second argument of the comparison into a new **let** clause. Here, we treat the comparison as a boolean function.

```
for $p in distinct −values(doc("bib.xml")// publisher )
return
  <publisher>
        <name> { $p } </name>,
        { for $b in doc("bib.xml")//book[$p eq publisher ]
          return $b/ title
        }
  </publisher>
```

$\overset{N-3.6}{\rightarrow}$ (move **return** clause into a new **let** clause)

```
for $p in distinct −values(doc("bib.xml")// publisher )
let $v1 :=
   <publisher>
        <name> { $p } </name>,
        { for $b in doc("bib.xml")//book[$p eq publisher ]
          return $b/ title
        }
  </publisher>
return $v1
```

$\overset{N-3.8}{\rightarrow}$ (move the arguments of the element constructor into a new **let** clause)

```
for $p in distinct −values(doc("bib.xml")// publisher )
let $v2 := ( <name> { $p } </name>,
             for $b in doc("bib.xml")//book[$p eq publisher ]
              return $b/ title )
let $v1 := <publisher> { $v2 } </publisher>
return $v1
```

$\overset{N-3.7}{\rightarrow}$ (bind the complex expressions in the sequence of expressions to new **let** clauses)

```
for $p in distinct −values(doc("bib.xml")// publisher )
let $v4 := (for $b in doc("bib.xml")//book[$p eq publisher ]
            return $b/ title )
let $v3 := <name> { $p } </name>
let $v2 := ( $v3, $v4 )
let $v1 := <publisher> { $v2 } </publisher>
return $v1
```

$\overset{N-3.6}{\rightarrow}$ (move **return** clause into a new **let** clause)

```
for $p in distinct −values(doc("bib.xml")// publisher )
let $v4 := (for $b in doc("bib.xml")//book[$p eq publisher ]
            let $v5 := $b/ title
            return $v5)
let $v3 := <name> { $p } </name>
let $v2 := ( $v3, $v4 )
let $v1 := <publisher> { $v2 } </publisher>
return $v1
```

$\overset{N-3.10}{\rightarrow}$ (turn an XPath predicate into a **where** clause)

```
for $p in distinct −values(doc("bib.xml")// publisher )
let $v4 := (for $b in doc("bib.xml")//book
            let $v5 := $b/ title
            where $p eq $b/ publisher
            return $v5)
let $v3 := <name> { $p } </name>
let $v2 := ( $v3, $v4 )
let $v1 := <publisher> { $v2 } </publisher>
return $v1
```

$\overset{N-3.8}{\rightarrow}$ (move the complex arguments of the value comparison into a new **let** clause)

```
for $p in distinct −values(doc("bib.xml")// publisher )
let $v4 := (for $b in doc("bib.xml")//book
            let $v5 := $b/ title
            let $v6 := $b/ publisher
            where $p eq $v6
            return $v5)
let $v3 := <name> { $p } </name>
let $v2 := ( $v3, $v4 )
let $v1 := <publisher> { $v2 } </publisher>
return $v1
```

Let us examine the structure of the normalized query. We observe:

1. All data retrieval is done in the **for** or **let** clauses.

2. All **return** clauses only contain variable references.

3. Function calls, value comparisons, or element constructors only refer to variables. They do not contain complex expressions as arguments.

Thus, we have established the desired normal form. Notice that we reach the same normal form when grouping is expressed using a **let** clause. In this case, we would save the first rewriting step (N-3.6). Notice that the rewrites are only correct because element construction does not depend on namespace declarations.

Detecting Join Predicates

The following example query contains a nested query with aggregate function `fn:min`. The value of this function depends on the items computed by a correlated path expression. We would like to extract the correlation predicate in the path expression to be able to optimize this query.

We proceed as follows: The first three normalization steps replace complex expressions in the **return** clause or in function calls by variable references and introduce new **let** clauses (N-3.6). Then, we break the path expression into two parts using rewrite N-3.11 so that the first path expression ends in a predicate. In the next step, we use this result to extract the predicate and move it into the **where** clause of a new FLWR expression (rewrite N-3.10). In the last normalization step, we use rewrite N-3.8 to extract the path expression from the predicate in this **where** clause.

```
let $doc := doc("prices.xml")
for $t in $doc//book/title
return
  <minprice>
    { min($doc//book[title eq $t]/price) }
  </minprice>
```

$\overset{N\text{–}3.6}{\rightarrow}$ (move **return** clause into a new **let** clause)

```
let $doc := doc("prices.xml")
for $t in $doc//book/title
let $v1 :=
  <minprice>
    { min($doc//book[title eq $t]/price) }
  </minprice>
return $v1
```

$\overset{N\text{–}3.8}{\rightarrow}$ (move the arguments of the element constructor into a new **let** clause)

```
let $doc := doc("prices.xml")
for $t in $doc//book/title
let $v2 := min($doc//book[title eq $t]/price)
let $v1 := <minprice> { $v2 } </minprice>
return $v1
```

$\overset{N\text{–}3.8}{\rightarrow}$ (move the argument of function `fn:min` into a new **let** clause)

```
let $doc := doc("prices.xml")
for $t in $doc//book/title
let $v3 := $doc//book[title eq $t]/price
let $v2 := min($v3)
let $v1 := <minprice> { $v2 } </minprice>
return $v1
```

$\overset{N\text{–}3.11}{\rightarrow}$ (split path expressions behind a predicate)

```
let $doc := doc("prices.xml")
for $t in $doc//book/title
let $v3 := (let $v4 := $doc//book[title eq $t]
            for $v5 in $v4/price
            return $v4)
let $v2 := min($v3)
let $v1 := <minprice> { $v2 } </minprice>
return $v1
```

$\overset{N-3.10}{\rightarrow}$ (turn an XPath predicate into a **where** clause)

```
let $doc := doc("prices.xml")
for $t in $doc//book/title
let $v3 := (let $v4 := (for $v6 in $doc//book
                        where $v6/title eq $t
                        return $v6)
            for $v5 in $v4/price
            return $v4)
let $v2 := min($v3)
let $v1 := <minprice> { $v2 } </minprice>
return $v1
```

$\overset{N-3.8}{\rightarrow}$ (move the complex argument of the value comparison into a new **let** clause)

```
let $doc := doc("prices.xml")
for $t in $doc//book/title
let $v3 := (let $v4 := (for $v6 in $doc//book
                        let $v7 := $v6/title
                        where $v7 eq $t
                        return $v6)
            for $v5 in $v4/price
            return $v4)
let $v2 := min($v3)
let $v1 := <minprice> { $v2 } </minprice>
return $v1
```

Obviously, this sequence of rewrites achives its goal: All retrieval is done in **for** or **let** clauses, we have broken up complex expressions. Looking at the result of the normalization steps, it is evident that the XPath predicate has become a predicate in the **where** clause. Now, the corelation predicate is much easier to detect during query unnesting.

Nested Path Expressions

Predicates in path expressions might contain further nested path expressions. A nested path expression inside the filter expression of another path expression is (in most cases) semantically equivalent to an existentially quantified query. This similarity is our motivation to break path expressions such that embedded predicates become visible. As we will see below, our normalization rewrites make sure that nested path expressions and existential quantifiers are treated in a uniform way. This substantially simplifies query optimization because we have to support fewer cases of nested queries.

In the example query below, the predicate of the path expression contains two path expressions. One depends on the previous location step, while the other is independent of the context. We want to avoid to evaluate the context-independent path expression for every context node. As we want to apply optimizations to these nested path expressions, we first apply rewrite N-3.11 and then rewrite N-3.10. In the final step, we turn the general comparison into quantified expressions (rewrite N-3.9).

```
let $d := doc("bib.xml")
for $t in $d//book[author = $d//book/editor ]/title
```

```
return $t
```

$\overset{N-3.11}{\rightarrow}$ (split path expressions behind a predicate)

```
let $d  := doc("bib.xml")
let $v1 := $d//book[author = $d//book/editor ]
for $t in $v1/title
return $t
```

$\overset{N-3.10}{\rightarrow}$ (turn an XPath predicate into a **where** clause)

```
let $d  := doc("bib.xml")
let $v1 := $d//book
for $t in $v1/title
where $v1/author = $d//book/editor
return $t
```

$\overset{N-3.9}{\rightarrow}$ (turn a general comparison into a value comparison)

```
let $d  := doc("bib.xml")
let $v1 := $d//book
for $t in $v1/title
where some $v2 in $v1/author
        let $v3 := fn:data($v2)
       satisfies
      some $v4 in $d//book/editor
        let $v5 := data($v4)
       satisfies  $v3 eq $v5
return $t
```

In Chapter 4, we present a general framework for optimizing such expressions. For the specific case of nested location paths in XPath 1.0, we refer to [BKHM06].

3.4.4. Restrictions

Several of our normalization rewrites are only valid under the assumption that certain information of the involved subexpressions will not be observed in the remainder of the query. This information includes node identity, local namespace declarations, and nondeterminism of XQuery expressions. Besides our normalizations, these issues rule out many other optimizations. But for many queries they do not cause any problems, and hence our normalizations will be valuable in many cases. Consequently, we ignore them in this work.

Node Construction and Node Identity

In some cases, common subexpressions cannot be factorized [CDF+04, BCF+07]. For example:

```
( <a/>, <a/> )
```

is not the same as

```
let $x := <a/>
return ( $x, $x )
```

because the first expression constructs two distinct XML element nodes, whereas the second returns two identical XML nodes. This problem occurs in all rewrites that introduce new **let** clauses containing expressions with constructors. Since most operations do not exploit node identity, this problem is rarely an issue. In most cases, node construction is only done to construct the final result which is returned to the user.

Namespaces

When moving expressions, we need to be careful because element constructors might introduce new namespaces. When we move expressions out of these scopes, e.g. by introducing a new **let** expression, we violate these scoping rules as shown in the following example taken from [FK04]:

```
declare namespace ns="uri1"

for $x in fn:doc("uri")/ns:a
where $x/ns:b eq 3
return
  <result xmlns:ns="uri2">
    {for $x in fn:doc("uri")/ns:a
     return $x/ns:b }
  </result>
```

When we apply our normalization rewrites as usual, the FLWOR expression bound to variable $v2 is evaluated using namespace `uri1` instead of `uri2`:

```
declare namespace ns="uri1"

for $x in fn:doc("uri")/ns:a
where $x/ns:b eq 3
let $v2 := (for $x in fn:doc("uri")/ns:a
            return $x/ns:b)
let $v1 := <result xmlns:ns="uri2"> { $v2 } </result>
return $v1
```

Thus, the rewrites might change the namespace declarations that are defined in the current evaluation context. In principle, one could establish the proper namespace declarations, but in this work we will ignore the problem of namespaces.

Ordering Mode

The result of the following expression is not deterministic. Depending on the order in which the values in the input sequence are applied to the predicate list, the result of this expression can either be an error or the value 3.

```
unordered{
  ("foo", "bar", 3)[ floor (.) < 5][1]
}
```

Hence, one must be careful when inferring unorderedness in subexpressions of queries, e.g.

```
some $i in ("foo", "bar", 3)[ floor (.) < 5][1]
 satisfies true
```

Again, we will ignore these issues in our optimizations and assume deterministic results. We refer to [GRT07] for a further discussion on this topic.

3.5. Translation into Logical Algebra

The result of normalization, discussed in the previous section, will now turn out to be a convenient starting point for the translation of XQuery queries into our algebra. Let us therefore summarize the structure of normalized queries as they are produced during normalization.

First, path expressions are broken up into simple path expressions. Consequently, we only need to treat simple path expressions without nested path expressions or predicates in

The binary $\mathcal{T}$ function for FLWOR expressions:

$$\mathcal{T}(Q, A) := \begin{cases} \mathcal{T}(\text{REST}, [tid_p(]\Upsilon_{x:\mathcal{T}_T(e)}(A))[]) & \text{if } Q = \textbf{for } \$x \text{ [}\textbf{at } \$p\text{] } \textbf{in } e \text{ REST or} \\ & \text{if } Q = \$x \textbf{ in } e \text{ REST} \\ \mathcal{T}(\text{REST}, \chi_{x:\mathcal{T}_T(e)}(A)) & \text{if } Q = \textbf{let } \$x := e \text{ REST and } e \text{ is sequence-valued} \\ \mathcal{T}(\text{REST}, \chi_{x:\mathcal{T}_I(e)}(A)) & \text{if } Q = \textbf{let } \$x := e \text{ REST and } e \text{ returns a single item} \\ \mathcal{T}(\text{REST}, \sigma_{\mathcal{T}_I(p)}(A)) & \text{if } Q = \textbf{where p REST} \\ \mathcal{T}(\text{REST}, Sort_{x_1 \ldots x_n}(A)) & \text{if } Q = \textbf{order by } \$x_1 \ldots \$x_n \text{ REST} \\ \Pi_e(A) & \text{if } Q = \textbf{return } \$e \\ A & \text{if } Q \text{ is empty string} \end{cases}$$

The unary functions $\mathcal{T}_T$ and $\mathcal{T}_I$ for other expressions:

$$\mathcal{T}_T(Q) := \begin{cases} \text{translation of [BKHM05].} & \text{if } Q \text{ is a simple path expression} \\ \Pi^D(\mathcal{T}_T(e)) & \text{if } Q = \texttt{distinct-values}(\text{e}) \\ \mathcal{T}(Q, \square) & \text{if } Q \text{ is a FLWOR expression} \\ \mathcal{T}_I(Q)\,[x] & \text{if } Q \text{ returns (a sequence of) items} \end{cases}$$

$$\mathcal{T}_I(Q) := \begin{cases} \exists t \in \mathcal{T}_T(R) : \mathcal{T}_I(P) & \text{if } Q = \textbf{some R satisfies P} \\ \forall t \in \mathcal{T}_T(R) : \mathcal{T}_I(P) & \text{if } Q = \textbf{every R satisfies P} \\ f(\mathcal{T}_I(e_1), \ldots, \mathcal{T}_I(e_n)) & \text{if } Q = f(e_1, \ldots, e_n) \\ v & \text{if } Q \text{ is a variable reference to variable \$v} \\ c & \text{if } Q \text{ is constant } c \end{cases}$$

Figure 3.4.: Translation of XQuery FLWOR expressions into the algebra

our translation function. Second, path expressions are only located in the **for** clause and the **let** clause. This assures uniform results after translation for different formulations of the same query. Third, nested query blocks are explicitly marked by FLWOR expressions or quantified expressions. Fourth, correlation between query blocks is explicitly handled in the **where** clause. Nested query blocks become subject to unnesting in later steps of the optimization.

3.5.1. Translation Function

Based on the properties mentioned above, we specify the translation procedure by means of three mutually recursive procedures $\mathcal{T}$ (see Figure 3.4). For a given query Q, $\mathcal{T}_T(Q)$ translates Q into our algebra.

The binary function $\mathcal{T}(Q, A)$ is responsible for translating a FLWOR expression Q into the algebra. The first argument of this function is the (remainder of) the query to be translated, and the second argument is the algebraic expression constructed so far. The result of each translation step is a tree of algebraic operators which produce sequences of tuples. For each clause of the FLWOR expression, we give the corresponding translation rule.

For non-FLWOR expressions, we use two different unary translation functions. Function $\mathcal{T}_I(Q)$ translates a subexpression Q into a function with a simple return type in the XQuery data model, while function $\mathcal{T}_T(Q)$ returns an algebraic expression which produces sequences of tuples. Notice that we rely on the translation presented in [BKHM05] to translate simple path expressions. However, in contrast to that proposal, we show in Section 3.6 that we do not fix the implementation of the location steps during translation. This decision is made during cost-based optimization instead.

Since a FLWOR expression can occur within simple expressions and vice versa, these functions are mutually recursive. In the translation rule for the **let** clause we explicitly select the translation function to use: if the expression bound in the **let** clause is sequence-valued, this sequence is turned into a sequence of tuples. Otherwise, we use the tranlsation function that returns single items.

3.5.2. Example

Let us consider the last example query of Section 3.4.3. Below, we repeat the result of normalization:

```
let $d := doc("bib.xml")
let $v1 := $d//book
for $t in $v1/ title
where some $v2 in $v1/author
        let $v3 := fn:data($v2)
       satisfies
      some $v4 in $d//book/ editor
        let $v5 := data($v4)
       satisfies $v3 eq $v5
return $t
```

We begin with the first **let** clause of the FLWOR expression. The translation results in:

$$\chi_{d:\mathcal{T}(\text{doc("bib.xml")})}(\Box)$$

After translating the function call in the subscript, we encounter another **let** clause.

$$\chi_{v1:\mathcal{T}(\$\text{d//book})}(\chi_{d:\text{doc("bib.xml")}}(\Box))$$

We continue with the **for** clause which is mapped to an unnestmap operator by the translation function.

$$\Upsilon_{t:\mathcal{T}(\$\text{v1/title})}(\chi_{v1:\Upsilon_{b:d//book}(\Box)}(\chi_{d:\text{doc("bib.xml")}}(\Box)))$$

The **where** clause is translated into a selection operator. We have to translate the predicate recursively.

$$\sigma_{\mathcal{T}(\ldots)}(\Upsilon_{t:\Upsilon_{tt:v1/title}(\Box)}(\chi_{v1:\Upsilon_{b:d//book}(\Box)}(\chi_{d:\text{doc("bib.xml")}}(\Box))))$$

We continue with the first quantified expression.

$$\sigma_{\exists x\in\mathcal{T}(\ldots):\mathcal{T}(\ldots)}(\Upsilon_{t:\Upsilon_{tt:v1/title}(\Box)}(\chi_{v1:\Upsilon_{b:d//book}(\Box)}(\chi_{d:\text{doc("bib.xml")}}(\Box))))$$

To avoid clutter, we will refer to the result of translating the range expression of the first quantified expression by e_1 and to the result of translating the range predicate of this existential quantifier by e_2. Thus, we get:

$$\sigma_{\exists x\in e_1:e_2}(\Upsilon_{t:\Upsilon_{tt:v1/title}(\Box)}(\chi_{v1:\Upsilon_{b:d//book}(\Box)}(\chi_{d:\text{doc("bib.xml")}}(\Box))))$$

The recursive translation of the range expression is similar to the translation of the **for** clause and **let** clause. The translation of the second quantified expression is also similar to the translation of the first quantifier:

$$\begin{aligned} e_1 &:= \chi_{v2:\text{fn:data(v1)}}(\Upsilon_{v1:\Upsilon_{a:v1/author}(\Box)}(\Box)) \\ e_2 &:= \exists y \in \chi_{v4:\text{fn:data(v3)}}(\Upsilon_{v3:\Upsilon_{c:c/editor}(\Upsilon_{c:d/book}(\Box))}(\Box)) : v2 = v4 \end{aligned}$$

The last translation step consists of translating the **return** clause which introduces a projection.

$$\Pi_t(\sigma_{\exists x\in e_1:e_2}(\Upsilon_{t:\Upsilon_{tt:v1/title}(\Box)}(\chi_{v1:\Upsilon_{b:d//book}(\Box)}(\chi_{d:\text{doc("bib.xml")}}(\Box)))))$$

Clearly, the translation is a simple mapping of the normalized XQuery expression into our algebra. The resulting algebraic expression contains nested algebraic expressions – in this example existential quantifiers. One of our goals in the remainder of this work is to replace these nested algebraic expressions by algebraic operators for which more efficient evaluation techniques exist and which allow for more optimizations.

3.6. Query Representation

In the previous sections, we have presented our normalization steps as rewrites on the abstract syntax tree of the parsed XQuery query. We have also defined a translation function that maps XQuery constructs into our algebra. In this section, we discuss the implementation of the normalization and translation steps.

In our implementation, we integrate normalization, translation, and factorization of common subexpressions. We also assign a type to all translated constructs and annotate them with cost and cardinality information. Therefore, we use the schema component which we discuss in Section 3.7.

Most importantly, we do not translate the parsed query directly into an algebraic expression. Instead, our internal query representation unifies features of calculus and algebra. In this section, we give details on the mapping into our query representation. It is similar to the query graph model [HFLP89, JK84, SKS+01]. We will argue that for all expressions in our algebra, there is an equivalent representation in our query representation.

3.6.1. General Concepts

After the translation step, all steps of our optimizer work on a common query representation. As already mentioned, it is closer to a calculus representation used during the first heuristic rewrite phase. During cost-based optimization, it is turned into a representation closer to an algebraic expression annotated with implementation hints. Figure 3.5 contains the classes most relevant for this thesis. Next, we briefly survey the most important classes. Then, we introduce the important concepts that underlie our internal representation.

Class Hierarchy of the Query Representation

All classes of the query representation derive from the abstract base class `Expression`. Algebraic operators are represented by classes derived from the abstract class `Algebra` which in turn is a direct subclass of the class `Expression`. While algebraic operators consume and produce sequences of tuples, the remaining subclasses mostly return atomic values. Instances of expressions are often bound to the subscript of an algebraic operator. For example comparisons are handled by function calls (class `ExprFFunCall`), and these boolean functions can be connected by boolean operators handled by classes derived from class `ExprBool`. To facilitate the factorization of common subexpressions, we employ the class `IU` (Information Unit, see below). Briefly, an IU represents an (intermediate) result of an expression. For example, an IU might represent the value of an attribute inside a tuple returned as query result, or the intermediate result in an arithmetic expression. We already pointed out that our translation of simple path expressions does not decide how location steps are implemented. Thus, until this decision is made by the cost-based optimizer, we represent location steps by the class `ExprStep`. In Section 3.6.4, we give more details on our representation of path expressions.

The calculus flavour of our internal representation is realized by classes we call *block*s. Blocks are special classes because, apart from being `Expression`s, they also inherit from class `BlockMixIn`. This class injects the properties of a calculus expression to these classes. Currently, we use two blocks: `AlgSFWD` blocks represent FLWOR expressions and quantified expressions, and `AlgGroup` blocks express grouping explicitly and duplicate elimination (as a special case of grouping). We give more details later in this section.

Finally, there is a large number of algebraic operators. The most important unary operators are:

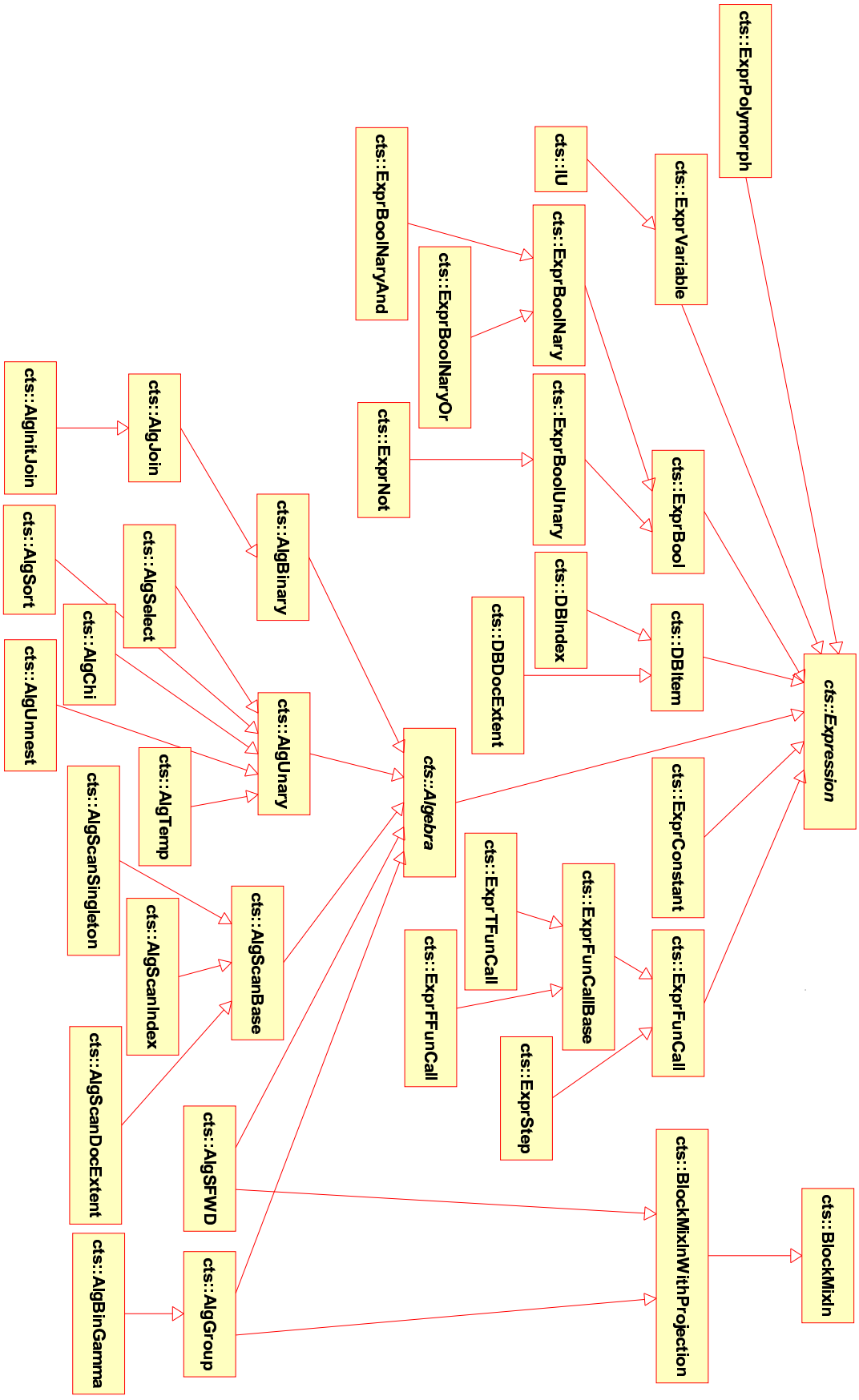


Figure 3.5.: Class Hierarchy of the internal query representation

`AlgSort`	is used for explicit sorting requests in an algebraic expression.
`AlgSelect`	filters tuples based on the result of evaluating the predicate p in its subscript.
`AlgChi`	represents the map operator χ.
`AlgUnnest`	represents the unnest map operator Υ.

Among the binary operators, the class `AlgJoin` is the most important one. It is used for all join types but the outer join and the d-join. The latter two join types are handled by the class `AlgJoinInit` because they both require an additional expression to initialize tuples.

Operations on the Query Representation

The class hierarchy of our query representation implements the *composite* pattern [GHJV95]. The composition models the argument relationship among the operators and expressions in the query. Additionally, the classes of our query representation mostly serve as information containers.

While they still contain some (recursive) functions to synthesize their values, most operations are performed by *visitors* or *mutators* [GHJV95]. This way, it is possible to add new operations to the classes of the query representation without actually touching their code.

A *visitor* only collects information during the traversal of the object structure. In addition, a *mutator* may change the visited elements. In our implementation, a *mutator* traverses the object structure from its root to its leaves and ascends back from the leaves to the root. Information can be collected during descending, while elements are only changed while ascending. All rewrites we present in this thesis are implemented as mutators. Several of them use visitors to collect information. This means that we implement all transformations explicitly in the visit operations of the mutators.

One might argue that a more declarative way of describing transformations on the query representation is desirable. We could specify rules declaratively when we formulate them as transformations in an attribute grammar [FMS93, ALSU06]. Consequently, we need to interpret our query representation as a type-annotated tree.

Information Units

During query execution, it is desirable, not to compute the same subexpression repeatedly. Hence, we have to detect common subexpressions in a query and explicitly share them. During rewriting, this sharing of subexpressions must stay intact.

We solve this problem by introducing information units into our query representation (class `IU`). Every instance of this class represents an abstract value at query evaluation time. Notice that *every* intermediate result is bound to an information unit, no matter if its result is actually shared. Currently, we detect and share common subexpressions in subscripts of algebraic operators. However, sharing common algebraic subexpressions is certainly a desirable future extension [Neu05].

Consider as an example the predicate `[position() ne 2 and position() lt 5]`. Obviously, the position of the context node is used twice in this predicate. We avoid to evaluate and represent the result of this function call by factoring its result as shown in Fig. 3.6. Notice that function `fn:position` has one implicit parameter which is the context position from the dynamic context. In this example, we assume that this context position is bound to a *named IU*, i.e. one can refer to the information unit by its name "cp". All other information units are not associated with a name, denoted by "-".

In the following sections, we describe how we use the query representation as a target of the translation step. We will focus on FLWOR expressions, quantified expressions and simple location paths, as they are most relevant to the optimizations we deal with in this thesis.

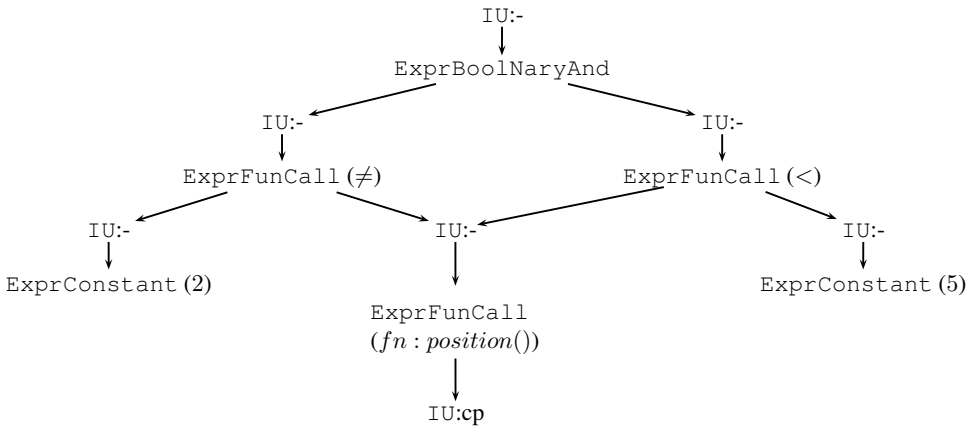


Figure 3.6.: Sharing of common subexpressions

3.6.2. FLWOR-Expressions

The translation presented in Section 3.5 yields a canonical operator tree as it is usually presented in database text books [GMUW01, RG02]. However, detecting patterns on such an algebraic expression is difficult and inefficient because the argument relationship is directly encoded into the algebraic expression. For many rewrites the exact argument relationship is not important. Such rewrites are more difficult to implement on algebra trees because pattern matching must consider more combinations of argument relationships.

Blocks

A query representation that is closer to a calculus representation simplifies pattern matching during heuristic rewriting. All classes, we consider as blocks, derive by multiple inheritance from class `BlockMixin` which introduces a class hierarchy independent of expressions. This class injects the concept of a *block* into the algebraic representation in the rest of the class hierarchy [BC90, SB98].

Figure 3.7 introduces some notation which we will subsequently use to refer to components of blocks. All blocks inherit from class `BlockMixIn`: a list of producers, P, similar to generators in a calculus expression, a list of `AlgChi` operators, C, each of which encapsulates the computation of an expression, a list of `AlgUnnest` operators, U, each of which represents the computation of a sequence-valued function whose result is immediately flattened, and a pointer to the parent block. Additionally, classes derived from class `BlockMixInWithProjection` contain a projection list, i.e. they implement Π_A for a set of attributes A. We have two different kinds of blocks.

Class `AlgSFWD` is derived from class `BlockMixInWithProjection` and extends this class with a predicate p as defined in Figure 3.7. First, we assume for simplicity only a FLWOR expression without **let** or **order by** clauses and with path expressions whose predicates are all moved into the **where** clause if possible. Then the semantics of this class is defined as the algebraic expression

$$\Pi_A(\sigma_p(\Upsilon_{p_n}(\ldots(\Upsilon_{p_2}(\Upsilon_{p_1}(\Box)))))).$$

Thus, the variable in the **return** clause constitutes the projection of the block. The **where** clause is represented by a selection operator, and the **for** clauses are implemented by a sequence of unnest map operators. When the producers p_i can be evaluated independently, we can turn the unnest map operators into cross products (via Eqvs. 2.17 and 2.18).

Notation	Description
Π_A	the attributes A specified in the final projection (class `AlgBlockMixinWithProjection`)
$P = \langle p_1 \dots p_k \rangle$	the producers P
$p = l_1 \wedge l_2 \wedge \dots \wedge l_m$	a conjunctive predicate (only in classes derived from blocks)
$C = \langle c_1 \dots c_n \rangle$	expressions c_i whose result is bound to a variable
$U = \langle u_1 \dots u_o \rangle$	sequence-valued expressions u_i whose result must be iterated over
$G = \langle g_1, \dots g_p \rangle$	grouping attributes

Figure 3.7.: Notation used for blocks

Remember that we also denote the concatenation of the application of algebraic operators with $\circ$. The semantics of this class is defined by the algebraic expression

$$\begin{aligned} &\Pi_A^{(i_1)} \circ \\ &\quad \sigma_{l_1}^{(i_2)} \circ \sigma_{l_2}^{(i_3)} \circ \dots \circ \sigma_{l_k}^{(i_j)} \circ \\ &\qquad \chi_{c_{\vec{n}}}^{(i_{j+1})} \circ \chi_{c_{\hat{n}}}^{(i_{j+2})} \circ \dots \circ \chi_{c_{\dot{n}}}^{(i_k)} \circ \\ &\qquad\quad \Upsilon_{p_1}^{(i_{k+1})} \circ \Upsilon_{p_2}^{(i_{k+2})} \circ \dots \circ \Upsilon_{p_n}^{(i_{l-1})} \circ \\ &\qquad\qquad \Box^{(i_l)}. \end{aligned} \tag{3.12}$$

The superscript (i_x) denotes the permutation of these operators that is consistent with the given XQuery expression. For every query, we have $(i_1) = 1$ and $(i_l) = l$, i.e. the projection is the outer-most operator and the singleton scan is the inner-most operator of this expression. The list `theApplicationOrder` represents this permutation of operators that maps positions in the resulting algebraic expressions to pointers of the operator at this position. Thus, after translation, the order of the entries in this list is consistent with the occurence in the textual query representation. This is too restrictive because a partial order of the expressions would suffice. But later rewrites can simplify these order constraints.

Notice that not all map operators in the define list of the block actually need to be part of the algebraic expression 3.12. Only the map operators resulting from the translation of **let** clauses are part of this expression. The remaining map operators in the define list wrap complex expressions.

Similarly, none of the unnest map operators of the unnest list appears in the algebraic expression 3.12. This list is only used to simplify detecting sequence-valued functions contained in any part of the FLWOR expression. But only the variable bindings resulting from these expressions belong to the result of this expression.

Now that we have introduced the block structure and the semantics of the SFWDU block implemented by class `AlgSFWD`, it is easy to map the FLWOR expressions to SFWDU blocks. The name of the SFWDU block is motivated by the main components of an SQL block (and also OQL block): **S**elect, **F**rom, **W**here, and auxiliary components **D**efine and **U**nnest. Each clause of a FLWOR expression is mapped to the components of the SFWDU block, as indicated by the solid arrows in Figure 3.8. The dotted lines pointing from the U- and D-component to the F-component represent the access to document nodes in path expressions in the **for** or **let** clause. The list of producers obtained this way provides a convenient access to schema information used to type the query. Notice that the **return** clause after normalization contains only a single variable binding. This projection is represented by a projection list in the **S**elect component. The **order by** clause is implemented by an sort operator (`AlgSort`) that consumes the result of the SFWDU block. Hence, the SFWDU block must not project attributes that are needed by the sort operator. Since this operator

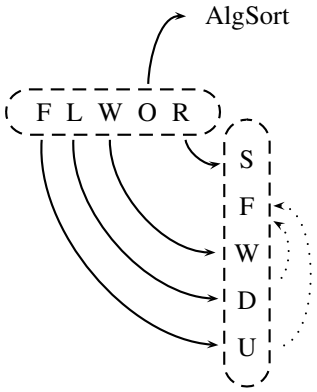


Figure 3.8.: Mapping of FLWOR expressions to the SFWDU block

can project away attributes, it establishes the required tuple signature.

Class `AlgGroup` is also derived from class `BlockMixInWithProjection` and extends it with a *grouping specification* and a predicate p, as defined in Figure 3.7. The grouping specification is a list of grouping attributes G. Let a be the elements in C that contain aggregate functions. The result of these expressions is bound to the set of attributes R. Then, the class `AlgGroup` implements the algebraic expression

$$\Pi_{R \cup G}(\sigma_p(\Gamma_{R;=G;a}(p_1))) \text{ for } \texttt{AlgGroup} \text{ and}$$
$$\sigma_p((p_1)\Gamma_{R;p_1.G=p_2.G;a}(p_2)) \text{ for } \texttt{AlgBinGamma}.$$

Notice that we assume that the producer list P contains exactly one entry for class `AlgGroup` and exactly two entries for class `AlgBinGamma`.

3.6.3. Quantified Queries

In our implementation, we translate universal quantifiers into negated existential quantifiers because of Eqv. 3.13

$$\begin{aligned} \forall x \in e : \neg(p) &= \neg\exists x \in e : p \\ &= \neg\exists x \in \sigma_p(e) : \text{true}. \end{aligned} \tag{3.13}$$

This decision is based on the observation that the canonical nested evaluation of a universal quantifier requires us to look at *all* tuples in e. After rewriting the universal quantifier into a negated existential quantifier, it is possible to stop after the first tuple that passes the negated predicate. Thus, in the average case, we can already stop after looking at half of the tuples.

As a consequence, when we look for universal quantifiers, we have to match a negated existential quantifier with predicate p modified accordingly. We map the existential quantifier in Eqv. 3.13 to an instance of class `ExprPolymorph`. Consequently, we map the universal quantifier to an instance of class `ExprNot` with an instance of class `ExprPolymorph` as argument. Moreover, notice that after applying Eqv. 3.13 we have moved a range predicate into the range expression. Thus, e contains the range predicate, **in** clauses of the quantified expression, and **let** clauses. Hence, it is natural to translate this range expression into a SFWDU block. We set a flag in the SFWDU block to mark it as *quantified*. This will be convenient when we rewrite quantified expressions.

3.6.4. Path Expressions

We have to pay special attention to path expressions for two reasons: (1) They are used in almost every XQuery query because they are needed to address data in an XML document. (2) We expect the optimization of path expressions to be especially critical for query performance. Thus, we now present the internal representation for path expressions.

Step Expressions

Consequently our main *requirements* for the internal representation of step expressions are that

1. Searching for complex patterns of location steps must be supported efficiently. Before we rewrite a path expression, we have to detect patterns. As this will be done quite often for a single query, efficient pattern matching on path expressions is a core requirement for our query optimizer.
2. Restructuring of path expressions must be fast. We expect that most optimizations of path expressions add or remove steps. Hence, this operation is especially performance critical.

Location steps are represented by the class `ExprStep`. Figure 3.9 shows the relevant parts of the class definition. We satisfy the requirements above by connecting simple path expressions in a doubly linked list. This means, that each step maintains a reference to its argument (Attribute `theNextStep` points to an `IU`, as all expressions do) and to its preceding step (attribute `thePreviousStep`). The doubly linked list is fast to modify, and it allows immediate traversal in both directions. Therby we assure that both pattern matching an transformations on path expressions are performed fast.

```
class ExprStep : public ExprFunCall {
  protected:
    // constructors and destructor
  public:
    // getter and setter functions
    float    cost ();
    float    cardinality ();
  private:
    Expression*          theNextStep;
    Expression*          thePreviousStep ;
    ExprAxisType         theAxis;
    ExprNodeKindTest     theNodeKindTest;
    natix :: QName       theNameTest;
    NLS::SchemaElementHandle theType;
};
```

Figure 3.9.: Definition of class `ExprStep`

The remaining members of class `ExprStep` specify the axis of the step (`theAxis`), and the node test. Information about the node test include the members `theNodeKindTest` and `theNameTest` with their obvious semantics. If only the former is specified, then the name test contains a wildcard, and if only the latter is specified, then the kind test stores the principal node kind.

During translation, every step expression is wrapped into an algebraic operator of class `AlgUnnest` and stored in the U-component of the enclosing SFWDU-block. If another instance of class `ExprStep` with the same arguments exists in this list, a pointer to this object is returned. This way, we factorize common subexpressions. Notice however, that this

is a purely syntactic check and, hence, we miss oportunities for factorization of common subexpressions that may be an equivalence test [AYCLS01, MS02].

Notice that the argument relationship in this internal representation reverses the order of path expressions. A step has its preceding step as argument.

Function Calls and Predicates

We do not need any special treatment for function calls. Inside path expressions, they are referenced as regular arguments and translated into instances of class `ExprFunCallBase`. In addition, every function call is wrapped into an algebraic operator of class `AlgChi` and stored in the D-component of the enclosing SFWDU block. If the same function call is already contained in this list, a reference to this function is returned. Thereby, we implement factorization of common subexpressions. Note that, as discussed in Section 3.4.4, factorization of common subexpressions can cause problems.

During normalization, we move XPath predicates into the **where** clause of a FLWOR expression. We add these predicates to the predicate of the enclosing SFWDU block when the path expression occurs inside a **for** clause. The list, `theApplicationOrder` in the SFWDU block takes care of the propor evaluation order. This is especially important in the presence of positional predicates.

Static and Dynamic Context

Every query in our query optimizer is wrapped into a SFWDU block. This simplifies the optimizer code in several places, but it also allows to add global information there. We use this top-level block to store information about the static context in the D-component.

Inside path expressions, the dynamic context is represented by three information units for context item, context position and context size. The latter two are only materialized if they are referenced anywhere. Thus, they are created on demand only. The context item is the IU that represents the result of an axis step. As a consequence, implicit references to the context are made explicit during translation.

3.6.5. Example

We resume our example from Section 3.4.3 and Section 3.5.2. After normalization, the query was as follows:

```
let  $d  := doc("bib.xml")
let  $v1 := $d//book
for  $t  in $v1/ title
where some $v2 in $v1/author
        let  $v3 := fn:data($v2)
        satisfies
       some $v4 in $d//book/editor
        let  $v5 := fn:data($v4)
        satisfies  $v3 eq $v5
return $t
```

The translation into our algebra resulted in the algebraic expression

$$\Pi_t(\sigma_{\exists x \in e_1 : e_2}(\Upsilon_{t:\Upsilon_{tt:v1/title}(\Box)}(\chi_{v1:\Upsilon_{b:d//book}(\Box)}(\chi_{d:\mathrm{doc("bib.xml")}}(\Box))))) \quad \text{with}$$

$$e_1 := \chi_{v2:\mathrm{fn:data(v1)}}(\Upsilon_{v1:\Upsilon_{a:v1/author}(\Box)}(\Box))$$

$$e_2 := \exists y \in \chi_{v4:\mathrm{fn:data(v3)}}(\Upsilon_{v3:\Upsilon_{c:c/editor}(\Upsilon_{c:d/book}(\Box))}(\Box)) : v2 = v4$$

In Figure 3.10, we show the internal representation for this query. Below the top-level block, the nested query blocks are clearly visible: An instance of class `ExprPolymorph` points from the second block to the third block, and another instance of this class refers

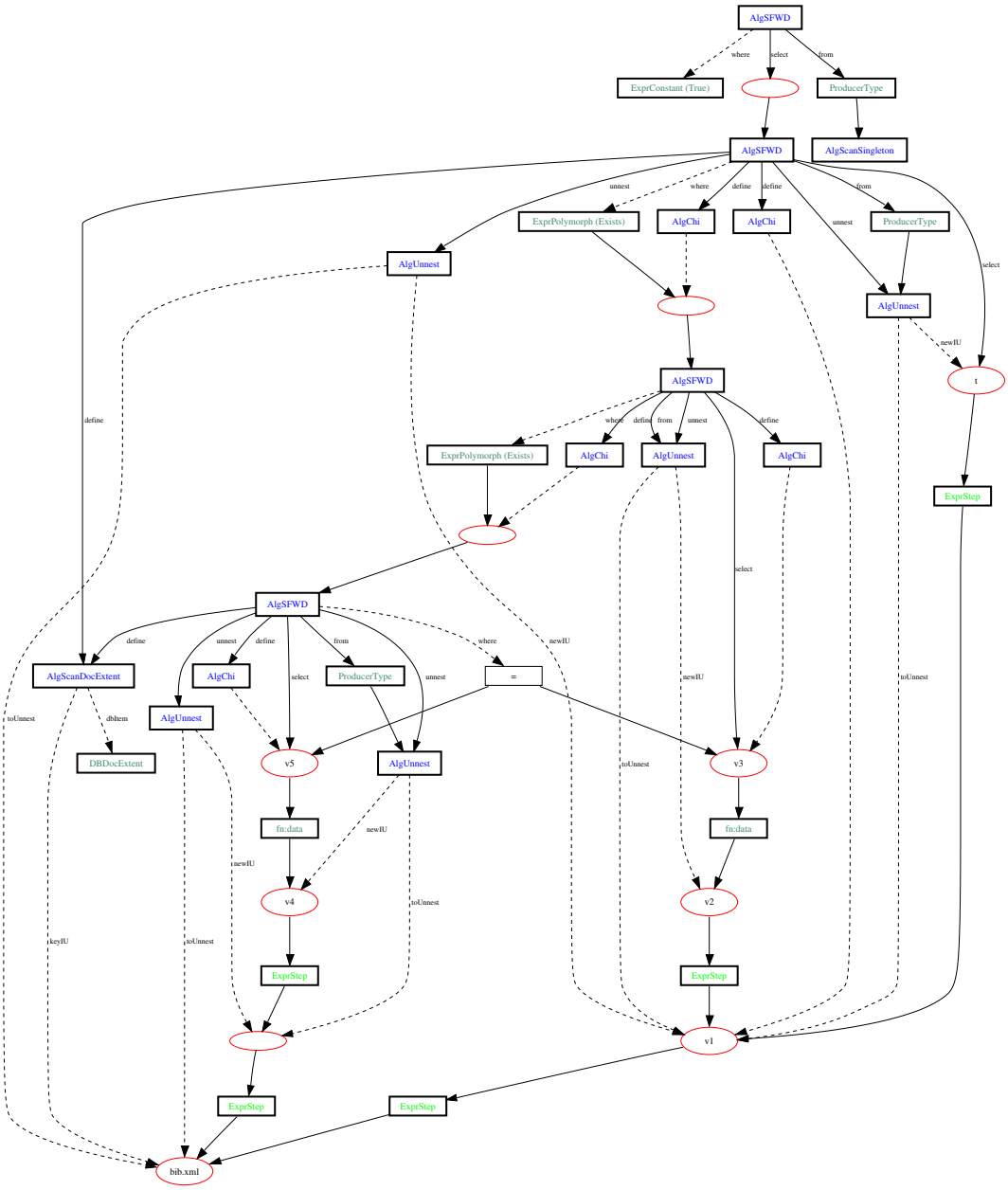


Figure 3.10.: Mapping of algebraic expressions to the Internal Representation

to the inner most query block. Also easy to detect are the path expressions. Two location paths end in the corelation predicate in the innermost query block. This corelation predicate compares IU "v3" with IU "v5". Both are computed by a sequence of axis steps represented by instances of class `ExprStep` followed by a call to function `fn:data`. Location steps are wrapped into `AlgUnnest` operators and also stored in the unnest list. Function calls and comparisons are wrapped into `AlgChi` operators and stored in the define list.

3.7. Typing

All optimization steps following the NFST phase assume a semantically correct query representation. If this assertion holds, all optimizations transform a valid query representation into an equivalent representation of the query. Thus, during the translation of the query it is necessary to check its structural and semantic correctness.

We have treated structural correctness in the previous three sections. In this section, we turn our attention to semantic correctness. In particular, we present the design of our schema management component. The check of semantic correctness includes:

- All expressions are well-typed.
- All function calls and variable references can be resolved to their definitions.
- Given a function call, the number and type of the actual arguments match some definition of the function.
- All explicit type casts can be handled by (usually built-in) conversion functions.
- All implicit type conversions can be resolved.

The XQuery specification [BCF+07, DFF+07] requires dynamic typing, i.e. performing type checks at runtime, and allows to do static typing at compile time. Hence, in any case, we to support the XQuery type system, which is based on XML Schema. In our system, we will support static typing for several reasons [CDF+04]. (1) In some cases, it can guarantee that given valid input data, the result of a query will obey to a desired output schema. (2) It can detect errors without executing the query. Thereby, it shortens the development cycle for XQuery-based applications by providing this useful debugging information. (3) Type information can be useful at query optimization time (e.g. [PMC02, KG02]). It might even be possible to integrate type information with XML synopsis to obtain more precise cardinality information.

We already have implemented the infrastructure for the schema component [Aly05], and its integration into the query optimizer is underway. Figure 3.11 shows its architecture. The schema component consists of the three subcomponents `Facade`, `Internal Representation`, and the `Life Cycle Management`, which we now discuss in detail.

3.7.1. The Schema Management Facade

The package `Facade` is the interface used by clients of the schema management. The query compiler, the query execution engine, or the storage system are the main users of this component. The query compiler uses it for static typing, the query execution engine for dynamic typing, and the storage system to validate persistent data.

The facade is split into two parts. The `SchemaMgr` and its derived class `XMLSchemaMgr` provide functions to load and store schemas and to query their content. The latter class contains functions specifically needed for working with XML schemas. Figure 3.11 shows only a part of the complete interface of these classes.

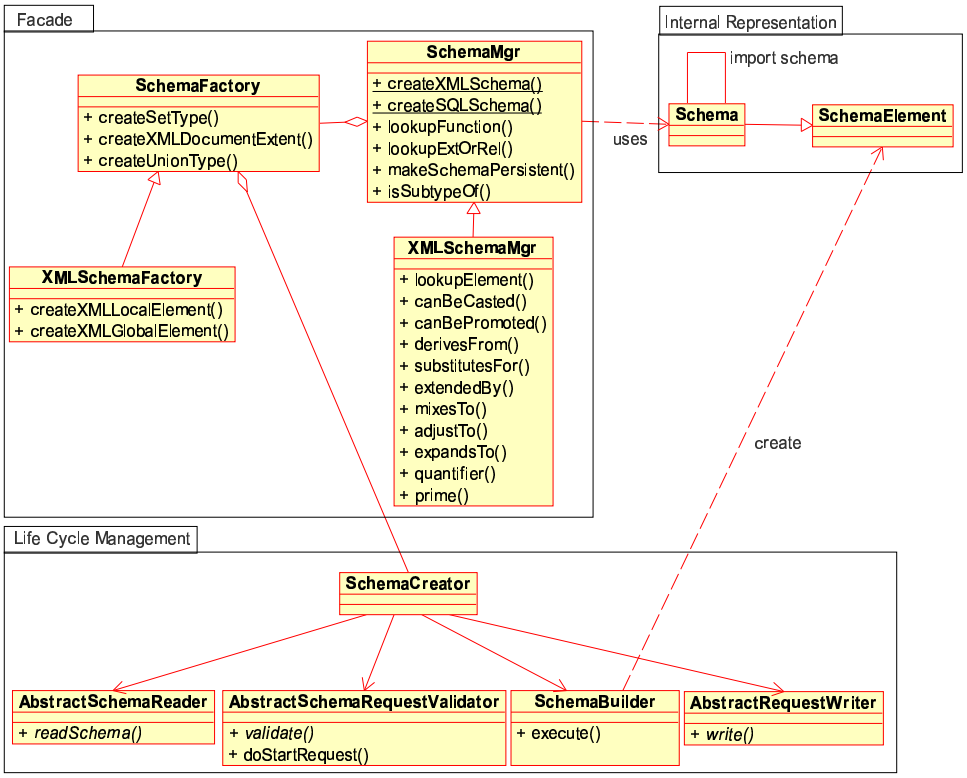


Figure 3.11.: Architecture of the schema management component

As detailed in [Aly05], the class `XMLSchemaMgr` implements all functions defined in the XQuery specification [DFF+07], including type inference, type containment checks, type casting and promotion, and typing of literals. For comparisons on content models, we transform the content models into DFAs and thereby map the type comparison to a comparison of DFAs, e.g. the structural equality of a type maps to the equality of the corresponding automata. The schema facade also serves as the entry point to information about the physical schema, i.e. the physical storage location or statistics for an XML document.

The schema facade hides the actual definition and implementation of the schema elements from the users of the schema management component. All these functions use only *handles* [Str00] of `SchemaElements` when referring to the content of a schema. This allows us to change the internal representation of schemas without affecting the clients of the schema management component.

The class `SchemaFactory` and its derived class `XMLSchemaFactory` implement factory methods [GHJV95] to construct types independent of the internal representation of types.

3.7.2. Internal Representation

The internal representation is implemented as a deep hierarchy of classes. This hierarchy is general enough to store schema information for different data models. For example, we successfully use the schema representation for both SQL and XML Schema. Common features are factored and used by several schema languages. At the same time, classes specific to XML Schema are completely separated. The classes in the `Facade` use and combine the information stored in the internal representation to satisfy requests by clients. However, we were careful to shield clients from the internal representation of a schema.

Since all classes in the internal representations are subclasses of class `SchemaElement`, it suffices to expose this base class as a *handle* [Str00] to the clients.

3.7.3. The Life Cycle of a Schema

The XQuery specification describes how the types defined in an XML Schema document are made available in an XQuery statement via the `import schema` statement. The location hint in this statement may or may not be used by the XQuery processor. In particular, we would like to be able to reuse a schema already loaded into memory. When bulkloading an XML document, we want to validate it and associate the XML Schema documents used for validation with the document. Then it is natural to make the XML Schema documents persistent inside Natix so that we can refer to them in validated documents or in XQuery statements.

Summarizing, we would like to load XML Schema information stored in different formats and in different locations, We also want to materialize schema information into a file on disk or inside the Natix system, e.g. we need to test type equality at runtime where one of the types is computed at compile time. Then it is desirable to simply reload the computed type information at query execution time.

We support this desired flexibility in the life cycle management of the schema management component. Let us. therefore, trace the life cycle of a schema: When a client refers to an XML Schema, it must be loaded into main memory first. Usually, this is done via the static method `SchemMgr::createXMLSchema`. This method forwards this request to the class `SchemaCreator`. When the required schema is already loaded into the internal cache, a reference to the schema is returned immediately. Otherwise, class `SchemaCreator` chooses a concrete implementation of class `AbstractSchemaReader` to read a schema file. One choice is to read a XML Schema file from disk, another is to read a materialized schema from the current Natix instance. The class `SchemaCreator` may also validate the loaded schema using a concrete subclass of class `AbstractSchemaRequestValidator`. Finally, the instance of class `SchemaBuilder` constructs the main memory representation of the schema.

When the client does not need the schema anymore, it destroys the `SchemaMgr`. This way it signals the schema management component that the schema can be removed from main memory. However, the `SchemaCreator` is free to overrule this request, e.g. the `SchemaCreator` might detect that the schema is still used by another client or it decides to cache the schema for later reuse.

Moreover, the client may ask the schema management component to materialize a schema. Again, class `SchemaCreator` chooses a concrete subclass of class `AbstractRequestWriter` to materialize the content of the current schema. The storage format of the materialized schema is specifically designed for a convenient materialization and loading of schema information. Notice that the life cycle management is largely independent of the internal representation of the schema. Creating `SchemaElements` is done via the `SchemaBuilder` who delegates this task to the `SchemaFactory`. The information needed to materialize and, later on, load schema information is basically independent of the internal representation of the schema.

3.7.4. Summary

Our schema management component provides all functions required to type XQuery statements. The internal representation of schemas and the mechanism to load and materialize schemas is easy to extend. We exploit this flexibility to load and store schema information of different data models (e.g. relational schema and XML Schema), different storage locations (e.g. text file or inside a Natix instance), and different storage formats (e.g. load XML Schema file or our native storage format). We refer to [Aly05] for details on the design of our schema management component and an assessment of the performance characteristics.

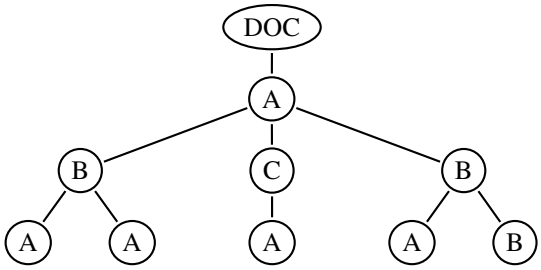


Figure 3.12.: An XML document

3.8. XPath Cardinality Estimation

Besides type information, every object of the query representation is annotated with cardinality information. The estimated result size for the query result as well as intermediate results are important information for cost-based query optimization. If statistics were gathered on queried data, we can compute precise cardinality estimates. Otherwise, we need to rely on simple heuristics.

We use well-known and well-studied techniques to estimate the cardinality of standard relational operators such as selections, joins, duplicate elimination, see [Ioa03] for a survey. Therefore, we concentrate on estimating the result size of path expressions in this section. Among the numerous proposals we decided to implement Markov Tables [AAN01, LWP+02]. However, the architecture we present here also allows to incorporate other XML synopsis structures, e.g.XSketch [PG02], Bloom Histogram [WJLY04], or XSeed [ZOAI06].

3.8.1. XML-Specific Challenges

Let us first review why XML cardinality estimation requires specific treatment. This analysis will directly lead us to the requirements. Consider the XML document depicted in Figure 3.12 as a rooted ordered labeled tree. The nodes in this tree represent XML elements.

First, notice that we deal with tree-structured data as opposed to flat tuples in the relational data model. In the tree, the path from the root to some element node can contribute information useful for estimating the number and kinds of elements to be found in the subtree rooted at this node. For example the node labeled “A” which is the only child of the node labeled “DOC” only contains children labeled “B” or “C” whereas all other nodes labeled “A” occur only as leaf nodes. Second, XML data might contain structural skew. Since elements can be optional, repeated, or chosen from a set of valid elements, the content model of nodes with the same label can be highly different, even when they are valid with respect to some schema. In Figure 3.12, for example, the content of the elements labeled “B” is different. Third, the result of queries may be sensitive to document order. This is the case, for example, when elements are filtered by a positional predicate.

3.8.2. Requirements

At best, we want estimates that are precise despite the challenges mentioned above. In particular, our requirements are:

1. Precise estimates for common query patterns,
2. Ability to trade precision for storage requirements,
3. Updateability,

4. Low construction overhead,

5. Fast and incremental computation of estimates.

They are not only desirable in the context of XML data [LWP+02] but also of relational data [PHIS96]. We want to have precise cardinality estimates because the cardinality is the most important parameter of cost functions. Inaccurate estimates lead to wrong decisions by the cost-based query optimizer and to poor query performance. However, the synopsis structures used to obtain the cardinality estimates should be small compared to the accessed data. The synopsis structures must fit into the available main-memory at optimization time and main memory is a scarce resource during optimization. The effort for creating a synopsis structure should be small so that it has only a small impact on the uptime or the performance of the database system during regular processing. Updateability assures low estimation errors even in the presence of updates of the base data. Finally, computing the estimates based on the synopsis should be efficient to do. When we estimate the result cardinality of a path expression, we are also interested in result sizes if intermediate results. As we will see in Chapter 5, this information may affect the decisions made by the query optimizer.

Given these requirements, we decided to implement the Markov Table [AAN01, LWP+02]. It can used to estimate the cardinality of simple path expression. Simple path expressions start with descendent step and may be followed by an arbitrary number of child steps possibly containing wildcards. For other axis steps, we base our estimates on simplifications and assumptions. Result cardinalities can be computed incrementally for every location step. This is not really possible for Bloom Histograms [WJLY04]. Intermediate result sizes are important information for the cost-based optimizer [HDN+03, MBB+06]. Especially compared to XML Synopsis [PG02, ZOAI06], the effort for construction is low. When updating the data store, it is possible to update the Markov Table without complete recomputation.

3.8.3. Architecture

Figure 3.13 presents the architecture of the XML statistics component. All concrete implementations of the XML cardinality estimators are accessed via the facade class `StatisticsMgr`.

Given a concrete estimator, e.g. an instance of class `MarkovEstimator`, statistics for an XML document can be gathered using the function `createFromFile`. The internal data structures for an estimator might be quite different, but this fact is hidden from the user. This specific data can be materialized or loaded into memory using functions `loadFromXMLFile` and `writeToXMLFile`. The `MarkovEstimator` is able to updates of the underlying data via function `update`.

For all XML estimators, cardinality estimation for a path expression works as follows. The method `getRootContext` returns an instance of class `EstimatorContext` that represents the document root node. We assume that even for variable references representing a nodeset, we have previously traversed some location path to estimate the cardinality of this nodeset. Hence, in this case and for all relative path expressions, the query optimizer or any other client of the statistics has an `EstimatorContext` that represents the context nodes for an axis step. If we want to know the result cardinality for some path expression, we then call function `traverseStep`. This function takes the `EstimatorContext` and information about the axis step as arguments. All these information are carried by an instance of class `Step`. With this information, the cardinality estimator computes a new `EstimatorContext` that represents the (virtual) result of the axis step. Clients may retrieve the result cardinality of the step by call function `getCardinality2` (function `getCardinality` returns the cardinality of the input to the location step). Notice that the same context can be used to traverse two different axis steps. This can be used to estimate the result cardinality even for branching path expressions. However, none of our estimators exploits correlations between the branches in the path expression yet.

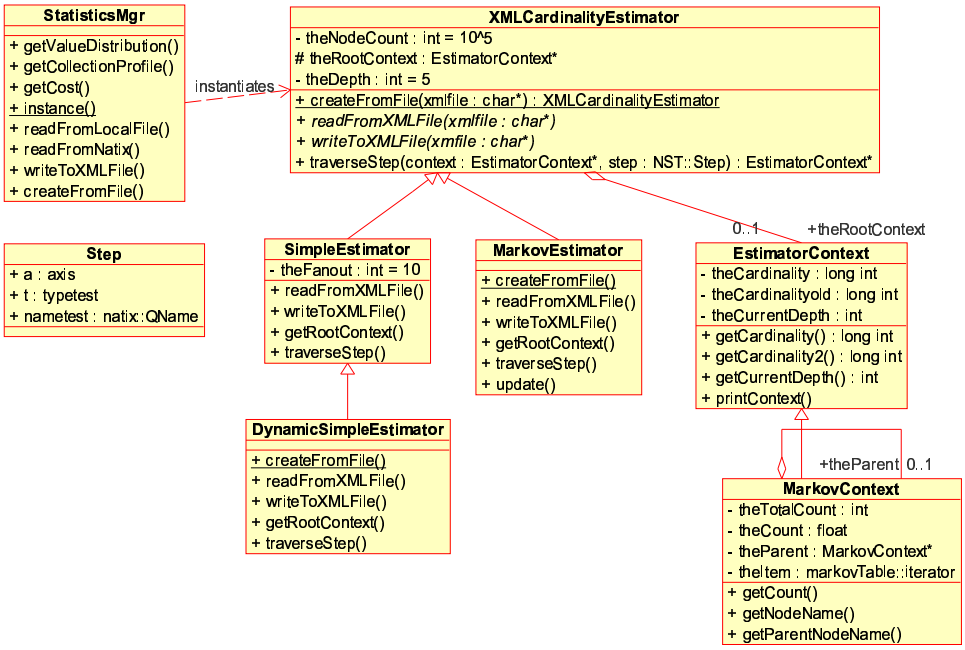


Figure 3.13.: Architecture of the statistics component

3.8.4. Simple Estimators

Among the three estimators we have implemented, class `SimpleEstimator` is the simplest. This estimator stores the number of nodes in the document (`theNodeCount`) and the average fan out of nodes in the document (`theFanout`). In this class, we use the standard values `theFanout=10`, `theDepth=5`, and the `theNodeCount`= 10^5, which seem to be reasonable estimates of real-world documents [MBV03]. The subclass `DynamicSimpleEstimator` reads these values in a single scan over the document.

Both classes share the simple estimation scheme shown in Figure 3.14. Independent of the name test of any axis step, we assume that every node has f children that qualify as query results. This assumption directly leads to the formulas in the first four rows in the figure. Considering the result cardinality of the sibling axes, we assume that each context node splits the children of its parents into two halves. Similar reasoning with respect to the whole document leads to the formulas for the preceding and following axis. The cardinality and depth for the remaining steps can be computed trivially with these formulas. In some cases, we can use the statistics to detect when we leave the document. For path expressions only containing parent, child, and sibling axes, we can determine that we move beyond the document root or the deepest leaf node. In any case, we can use the total count of nodes in the document to limit the result cardinality.

3.8.5. Markov Estimator

Using the `MarkovEstimator`, much more precise estimates can be computed, particularly when axis steps contain name tests. The table computed from the example document in Figure 3.12 is shown in Figure 3.15. Aboulnaga et al. [AAN01] argue that only little accuracy is lost when we restrict ourselves to a Markov process of order 1. Thus, we only keep paths of length 1 and 2 in the Markov Table and compute the cardinality of longer paths with the formula below. In this formula, $f(t_i, t_{i+1})$ refer to entries in the Markov Table that are pairs of labels denoting a parent-child relationship, and $f(t_i)$ to entries that

Axis	result cardinality	result depth
child and attribute	$c \cdot f$	$min(d+1, dt)$
descendant	$c \cdot f^{dt-d}$	-
parent	c^{-f}	$max(d-1, 0)$
ancestor	$c \cdot f^{-d}$	-
following-sibling, preceding-sibling	$0.5 \cdot c \cdot f - 1$	d
following, preceding	$ct - c \cdot f^{dt-d}$	-
self	c	d

with

ct	total number of nodes in the document
dt	depth of the document
f	average fanout of a node
c	cardinality of the context sequence
d	current depth of the context

Figure 3.14.: Cardinality estimation for simple estimators

contain the total number of occurrences for a tag name t_i.

$$\text{sel}(//t_1/t_2/\ldots(t_{n-1}/t_n) = \left(\prod_{i=1}^{n-2} \frac{f(t_i, t_{i+1})}{f(t_{i+1})}\right) \cdot f(t_{n-1}, t_n) \tag{3.14}$$

For example, the cardinality of the path expression `//A/B/A` is computed as follows.

$$\begin{aligned} \text{sel}(//\text{A}) &= f(A) = 5 \\ \text{sel}(//\text{A/B}) &= f(A, B) = 2 \\ \text{sel}(//\text{A/B/A}) &= \frac{f(A,B)}{f(B)} \cdot f(B, A) = \frac{2}{3} \cdot 3 = 2 \end{aligned}$$

Clearly, we can compute the result cardinality of the complete path incrementally. Thus, as a side effect, we also compute cardinalities for intermediate results. In this case, the cardinality of intermediate results is precise. But the cardinality of the final result is slightly underestimated because the Markov Table assumes structural uniformity and independence between paths. By looking at the document, it is clear that this assumption does not hold here.

parent	child	count
A	-	5
B	-	3
C	-	1
DOC	-	1
A	B	2
A	C	1
B	A	3
B	B	1
C	A	1
DOC	A	1

Figure 3.15.: Markov Table for the document in Fig. 3.12

To support the other axes, we need to rely on heuristics. We estimate the result cardinality of the descendant axis, the following axis, and the preceding axis with name test as half the cardinality of the overall occurrences of the given tag. Similar to the simple estimators, we estimate that the current node is in the middle of its parent and, thus, half of the siblings precede and follow the current node.

Notice that the Markov Table returns a specific `MarkovContext`. This context stores a pointer to the parent context. Thereby, we can precisely compute parent steps and ancestor with and without wildcards when we have reached the current context via child steps. Otherwise, we use the current (or total) depth, use the entries $f(t_i, t_{i+1})$ as estimates for the fanout of the parent node, and divide the current cardinality by this value.

	DBLP	XMark SF10	Swissprot	Treebank
Size	286 MB	1172.3 MB	115 MB	86 MB
# elements	6701980	16703210	2977031	2437666
parsing time	31s	66s	16s	7s
generate statistics	89s	237s	43s	31s
materialize statistics	0.01s	0.01s	0.01s	0.05s
load statistics	0.02s	0.02s	0.02s	0.12s
TreeSketch	2220s	-	DNF	DNF
XSeed	1620s	-	15660s	7620s

Figure 3.16.: Execution times to generate Markov Table

3.8.6. Experiments

Hardly any publication on XML cardinality estimation has reported construction times for their XML synopsis. We agree with Zhang et al. [ZOAI06] that construction time is a serious issue. Therefore, we present our performance numbers for constructing the Markov Table on the system described in Appendix A.3.

Figure 3.16 reports the elapsed times. We explicitly show the time to read and parse the XML documents from disk using SAX. The parsing time is included in the time to generate the statistics. We also report the time to materialize the generated statistics and to load the materialized statistics from an XML file on disk. Our experiments suggest that the statistics can be generated at a rate of approximately 3MB per second. Moreover, parsing the XML input, which is done at approximately 10MB per second, is not the bottleneck yet.

The execution times for the Markov Table are more than 100 times faster than the execution times reported in [ZOAI06] for XSeed (on a slightly less powerful system). Moreover, TreeSketch [PGI04] used to benchmark XSeed does not even finish to construct the statistics of Swissprot or Treebank within 24 hours! Neither system was tested with XMark with scale factor 10. But on an XMark document with scale factor 1, XSeed needed 162 seconds, and TreeSketch did not finish within 24 hours.

It remains open whether users are willing to invest the tremendous cost for creating the statistics for TreeSketch or XSeed. Comparing the predecessor of TreeSketch, XSketch [PG02], with the Markov Table, Polyzotis and Garofalakis obtained only better precision for memory budgets well below 50KB of available memory. Notice, however, that the Markov Table approach can adjust to query feedback [LWP+02]. Frequently queried path expressions can be integrated into the Markov Table to enhance precision with little additional overhead.

3.8.7. Summary

We have presented a generic framework for estimating result cardinalities of path expressions on XML documents. We believe that other XML synopsis, e.g. XML Synopsis or XSeed, can be easily integrated into the framework we have presented here. The concept of the estimator context is so flexible that all relevant context information can be stored there. Fortunately, the client code is independent of the concrete XML cardinality estimator used and, therefore, such extensions can be integrated seamlessly.

We also plan to incorporate value statistics into our XML statistics. While in [PG06], value statistics are created for every node, we will generate statistics only for selected parts of the document [LWP+02, BEH+06]. We do this for two reasons: (1) The storage overhead for e.g. a histogram is only a good investment if these values are actually queried. Similarly, for strings or text, a lot of memory is needed for the value statistics. (2) Even when schema information is available, and we know that a sequence of digits is actually used as a number, certain queries will not be able to use the number value because they

are compared to an untyped value or a string. In these cases, the statistics cannot be used even when the underlying data is accessed by a query. Consequently, we require the user to specify a simple path expression and the type for the content of the resulting nodes. Then, we compute statistics for numbers, strings or keywords, knowing that they will be actually used. The implementation of these value statistics will not be hard-coded but can be specified by the user.

3.9. Related Work

There are two main approaches to optimize a query. In the first one, the query is transformed into an internal representation that can be interpreted by the query evaluation system [AC75, WY76]. In this approach, called *interpretation*, there are only limited possibilities for optimizations.

Thus, the *translation* of a query into an internal representation is now the dominant technique in query processing. Relational algebra and relational calculus equivalences became prime targets for the translation of query languages because one can formally prove the equivalence of two expressions. It is then the task of query optimization to find equivalent expressions that can be evaluated more efficiently.

Normalization and Translation before XQuery

Ceri and Gottlob [CG85] translate a SQL query into an algebraic expression in two steps: The first step transforms the SQL syntax into a restricted one establishing a normalized syntax. This simplifies the translation, the second step, in which the restricted SQL syntax ist translated into the algebra. The authors argue that the resulting algebraic expression can be optimized and, thus, efficiency of the resulting algebraic expression is not an issue. Moreover, it is shown that their translation establishes a normal form because syntactically different queries are translated into the same algebraic expression.

Calculus representations are another representation into which SQL is translated [NPS91, vB87, FM95]. Fegaras [FM95, FM00] and von Bültzingsloewen [vB87] use a normal form established during their translation as the basis for further optimizations. Optimizing nested queries either containing quantifiers or aggregate functions are the prime subjects of research here [JK84, Bry89, vB90, Nak90, FM00].

In Starburst and DB2, a query is translated into the Query Graph Model (QGM) [HFLP89, PHH92, PLH97]. QGM is a custom representation similar to the calculus representations above. For SQL and, as we will see later, also for XQuery, a mapping of query constructs to the QGM is defined. Unfortunately, only informal descriptions of this mapping are publicly available. Heuristic optimizations, such as the decorrelation of nested queries and the merging of QGM blocks, are performed on this representations.

Translation of XPath 1.0

Path expressions represent an important fragment of XQuery. Many features of the XPath 1.0 standard have become part of the XQuery specification. Thus, translation, optimization, and evaluation techniques proposed for XPath 1.0 should carry over to path expressions in XQuery.

We make use of efficient translations of XPath expressions. Gottlob et al. [GKP02, GKP03b] observed that XPath expressions have an exponential worst-case run time when subexpressions are evaluated repeatedly. They propose to use memoization to avoid this redundant work.

Along the same line, Helmer et al. [HKM02] translate XPath location steps without positional predicates such that creating duplicates is avoided. These ideas were extended

in [HM03], where redundant sort operations are removed when the result of the path expression will still be in document order. Brantner et.al [BKHM05] were the first to present a complete translation procedure for XPath 1.0 into algebraic expressions. A complete translation of XPath into SQL statements is proposed in [Gru02]. In another approach to the processing of the child and descendent axis, XPath is translated into an automaton [LMP02].

Normalization and Translation of XQuery

The idea of Ceri and Gottlob [CG85] to normalize the full query syntax into a core language is also proposed in the XQuery specification [DFF+07]. The formal semantics of XQuery is defined in terms of this core language. Thus, an interpretative view of evaluating XQuery is taken in the formal specification. While some implementations of XQuery implement these semantics literally, it was soon clear that efficient XQuery processing demands a query representation that is easy to optimize.

The first proposal to normalize XQuery was proposed by Manolescu et al. [MFK01]. Their normalization rules prepare XQuery statements for the translation into SQL statements. Thus, all normalization rules work on the level of XQuery statements, remove nested FLOWR expressions, and establish some normal form that is not formally characterized.

Extending previous work [FM00], Fegaras [FLBC02] translates XQuery statements into monoid comprehensions. Rewrites establish a unique normal form to prepare subsequent optimizations. Monoid comprehensions allow for an elegant integration of different bulk types. Expressions in this calculus can be checked to preserve order or duplicates. Unfortunately, the cited work does not seem to exploit this fact.

Another translation algorithm for a subset of XQuery was proposed by Viglas et al. [VGD+02]. The presented algebra does not seem to preserve because XML documents are not treated as *ordered* labeled trees. A unique feature of this approach is its explicit management of context information. This context information can be exploited during query optimization.

The Timber system [JAKC+02] follows a different approach. Queries are translated into pattern trees defined in the logical tree algebra TAX [JLST02]. Optimizations are defined as rewrites on this tree algebra. All pattern trees in TAX can be mapped to algebraic operators in the physical algebra [SSKH+02].

These early proposals do not fully support the XQuery specification. Some of these efforts included a translation of XQuery into SQL [KKN03]. However, the MonetDB/-Pathfinder project is the approach that covers a large subset of XQuery. In this system, XML documents are represented in a pre-/post order encoding that maps a unique identifier to each node in the document [Gru02]. This allows to construct SQL queries that retrieve all nodes that satisfy a path expression. Later, this translation was extended to larger fragments of XQuery [GST04, GT04].

In the following, as the standardization process of XQuery converged, the focus shifted to a more complete coverage of the XQuery specification.

The XQuery engine of the BEA streaming XQuery engine [FHK+04] translates XQuery expressions into an internal expression representation. While this representation shares the ideas of the relational algebra, it is specifically designed to represent XQuery expressions. Both normalization and optimization are carried out as rewrites on this query representation.

A similar approach is taken by Galax [RSF06]. This system implements the normalization of the XQuery specification literally. Afterwards, the resulting XQuery Core expressions are translated into an extended algebra. Algebraic rewrites allow for optimizations.

Commercial relational database products also support XQuery to a varying extent. Microsoft SQL Server [PCS+05] and Oracle XML DB [LKA05] support fragments of the XQuery specification. Queries are translated into algebraic expressions and, if possible,

rewriting techniques of the relational optimizer are used for optimizations. The IBM DB2 product [NdL05] builds upon the System RX prototype [OCP+05, BCH+06]. The QGM query representation of DB2 was extended to be able to represent XQuery statements. Necessary extensions to the relational engine and XQuery specific optimizations are described in [BEH+06].

Our query representation is similar to the query graph model [HFLP89, JK84, SKS+01], which has proven very powerful in performing transformations on the query graph. In this chapter, we have described the normalization and translation into this representation. We have formally specified our normalization rules, the translation functions and the mapping between our algebra and our calculus-like query representation. As others have done before, our normalization rules simplify the translation into our internal representation. In parallel to normalization, translation, and semantic analysis, we factorize common subexpressions. The latter task is based on the ideas of dependency-based optimization [CM93]. In this process, new variables are introduced as a preparatory step for factorization [CD92].

Typing and Statistics

During semantic analysis, a type is associated with every expression. The XQuery specification gives some rules for typing XQuery expressions [DFF+07]. Aly [Aly05] describes the architecture for computing types and performing validation in our system. During typing and validation, schema information provided by an XMLSchema can be exploited. The complexity of type checking is treated in [Suc02, GKP03a, Seg03]. Given some schema or type information, several optimizations become available [FS98, KG02, ZO02, PMC02].

The XQuery typing rules are imprecise in deriving result cardinalities. For some expressions, more precise result cardinalities can be derived. At best, statistical information is available for the XML data to access. Unfortunately, synopsis structures used successfully for relational data [MCS88, Ioa03] are only of limited use for semistructured data because (1) the result cardinality of a location step depends on the context in which this location step is evaluated, and (2) the uniform distribution assumption may not hold. Hence, specific techniques for a result size estimation of XML queries were developed – see [Ram06] for a survey. We could identify five main approaches to XML cardinality estimation. First, histogram techniques were adopted for querying XML [FHR+02, Sar03b, WPJ02]. Second, the idea of a Markov process was used to treat context information [AAN01, LWP+02, BEH+06]. Third, specific synopses [GW97, PG02, PG06, ZOAI06] were developed that can trade space for better accuracy. XML Synopses can both represent context information and adjust to local violations of the uniform distribution assumption. Finally, the Bloom Histogram [WJLY04] and statistical learning techniques were used for cardinality estimation and even for cost estimation [ZHJ+05].

We have implemented the Markov Table proposed in [AAN01, LWP+02] for structural cardinality estimation. However, our architecture is general enough to support other estimators. We plan to include value statistics for path expressions specified by the user and query feedback to adjust to updates and query workloads.

4. Query Unnesting

The translation of a query into NAL, our algebra, yields a logical plan that may not lead to an efficient evaluation strategy. Thus, the query optimizer first applies transformations to the logical plan to improve it. These transformations are usually heuristics, i.e. each transformations is assumed to improve the quality of the logical plan. These transformations are not guided by cost information.

Among these heuristics, query unnesting is a particularly performance critical. Nested queries in XQuery require additional attention because many queries can only be formulated using nested query blocks. It is well-known that nested queries often need much longer to evaluate than unnested ones because (1) they require an evaluation in nested loops while for unnested queries more efficient evaluation algorithms become available, and (2) unnested queries allow the cost-based query optimizer to consider more plan alternatives. Naturally, this leads to an increased optimization time. But this is a good investment because, as we will see in this chapter, query execution times usually improve by several orders of magnitudes after unnesting a query. For this reason, we will concentrate on techniques for unnesting nested queries in XQuery. Other heuristics need to accompany query unnesting to exploit the full potential of algebraic optimization, and we mention several of them in this chapter.

First, we enumerate requirements for our unnesting equivalences and their implementation (Section 4.1). In Section 4.2, we introduce the three basic algebraic patterns we detect with our unnesting techniques and present motivating examples for each of them. We do not simply enumerate algebraic equivalences that turn a nested query into an unnested one. Instead, we organize all our unnesting equivalences together with supporting rewrites into decision trees. We devote one section to the algebraic equivalences and support rewrites for one pattern of the three basic patterns (Sections 4.3, 4.4, and 4.5). We apply our framework to example queries, and thereby explain our optimization approach and discuss the performance impact of unnesting. In Section 4.6, we present the implementation of our unnesting framework. Finally, we summarize our work in Section 4.7 and an discuss work related to ours in Section 4.8.

4.1. Requirements

As a fundamental requirement, we demand that our equivalences are correct. Given this prerequisite, our equivalences will only be useful when we can expect an performance gain after unnesting that exceeds the effort of applying our unnesting equivalences. This leads to the following requirements (see also e.g. [Che98]):

Correctness Rewrites transform the query plan from a valid plan into another valid plan.

Effectiveness The rewriting system chooses those rewrites that result in most efficient plans among all applicable rewrites.

Comprehensibility It must be easy to relate a rewrite rule to an equivalence. Scheduling of rewrite rules must be easy to tune or extend.

Extensibility The rewriting system must be easy to extend with new rules and traversal strategies.

Efficiency Rewriting should be reasonably fast. In particular we expect that the additional effort of rewriting is small when no rewrite can be applied.

In this chapter, it is our main goal to convey the idea of our unnesting framework. Thus, we refer to Appendix A.2 for the proofs of our equivalences. The structure of our decision trees will be the main tool to argue for the effectiveness of our unnesting framework. The last three requirements focus on design and implementation issues. Hence, we treat them in Section 4.6 where we present the implementation of our unnesting framework.

4.2. Algebraic Patterns

Our unnesting equivalences detect algebraic patterns containing algebraic expressions in subscripts of selections or map operators. In this section we identify and motivate these basic patterns.

4.2.1. Quantified Queries

XQuery contains primitives for expressing quantification in queries. A quantified expression begins with a quantifier (**some** for existential, **every** for universal quantification), followed by one or more **in**-clauses that are used to bind variables. We refer to the **in**-clauses as *range expressions*. After that we have the keyword **satisfies** and a test expression (or *range predicate*). Conceptually, the range predicate is evaluated for each combination of variable bindings. In the case of the quantifier **some**, the expression is true if at least one evaluation of the range predicate returns true, in the case of the quantifier **every**, all tests have to evaluate to true.

Let us reconsider the following example query which uses an (existentially) quantified expression in the **where** clause:

```
for  $t1  in  doc("bib.xml")//book/ title
where some $t2 in  doc("reviews.xml")// entry / title
        satisfies   $t1  eq  $t2
return  $t1
```

General comparisons in XQuery employ implicit existential quantification when comparing sequences. During normalization, we rewrite these implicit quantifications into explicit ones. Since quantification occurs quite frequently in XQuery queries, it is important to optimize these expressions by unnesting them. The previous example query can be formulated in terms of general comparisons:

```
for  $t1  in  doc("bib.xml")//book/ title
where  $t1  = doc("reviews.xml")// entry / title
return  $t1
```

The query with explicit quantification is translated into the following algebraic expression (we ignore the **return** clause, implicit function calls, and the result construction).

$$\sigma_{\exists t \in (e_2):t1=t2}(e_1)$$

with

$$\begin{aligned} e_1 &:= \Upsilon_{t1:doc(\text{"}bib.xml\text{"})//book/title}(\Box) \\ e_2 &:= \Upsilon_{t2:doc(\text{"}reviews.xml\text{"})//entry/title}(\Box) \end{aligned}$$

The example query contains the pattern that all existentially quantified queries in our unnesting procedure exhibit:

basic patterns for nested quantified queries

$$\sigma_{\exists x \in \sigma_p(e_2):q}(e_1)$$
$$\sigma_{\forall x \in \sigma_p(e_2):q}(e_1)$$

In our unnesting rules we identify several variations of these patterns for expression e_1, the range expression e_2, or the range predicate p. For each kind of these patterns we give an equivalent unnested expression.

4.2.2. Implicit Grouping

The term *implicit grouping* is motivated by the fact that grouping in XQuery must be formulated using nested queries. Explicit grouping implies an explicit grouping construct in the surface syntax of the query language.

In XQuery implicit grouping is frequently used to restructure input documents or to aggregate data using an aggregation function such as `sum`, `count`, or `avg`. The following example query groups book titles by publishers:

```
for $p in distinct −values(doc("bib.xml")// publisher)
return
  <publisher>
        <name> { $p } </name>,
        { for $b in doc("bib.xml")//book[$p eq publisher]
          return $b/ title
        }
  </publisher>
```

Here, grouping is expressed by a nested query in the **return** clause. Normalization results in an alternative style of expressing grouping, pulling up the nested part of the **return** into a **let** clause:

```
for $p in distinct −values(doc("bib.xml")// publisher)
let $t := (for $b in doc("bib.xml")//book
           let $p2 := $b/ publisher
           let $t2 := $b/ title
           where $p eq $p2
           return $t2)
let $np := <name> { $p } </name>
let $res := <publisher> { $t, $np } </publisher>
return $res
```

Translating the normalized query into an algebraic expression, we get (again ignoring implicit function calls and result construction):

$$\chi_{t:\Pi_{t2}(\sigma_{p=p2}(e_2))}(e_1)$$

where

$$\begin{aligned} e_1 &:= \Upsilon_{p:\Pi^D(doc("bib.xml")//publisher)}(\Box) \\ e_2 &:= \chi_{t2:b/title}(\chi_{p2:b/publisher}(\Upsilon_{b:doc("bib.xml")//book}(\Box))) \end{aligned}$$

The key component of the translation is that the **let** clause is translated into a χ operator with a subexpression in its subscript. We identify the

basic pattern for implicit grouping

$$\chi_{g:f(\sigma_p(e_2))}(e_1)$$

Similar to quantified queries, we identify several variations of this basic pattern. For each variation of the pattern containing a nested algebraic expression, we devise an equivalent unnested algebraic expression.

4.3. Existential Quantifiers

This section, as well as the following two sections on universal quantifiers and implicit grouping, is structured as follows. We start with a (small) motivating example and then discuss the general strategy for unnesting queries of this type. After that, we describe the concrete equivalences used for unnesting and some rules for support rewrites. Having covered the foundations, we then present detailed examples for unnesting, applying the rules and equivalences introduced before. In this context we also validate the effectiveness of our approach by showing performance figures for the example queries.

4.3.1. Motivating Example

As a motivating example for queries containing existential quantifiers, let us reconsider the query from Section 4.2 (where we want to find all books with at least one review):

```
for $t1 in doc("bib.xml")//book/ title
where some $t2 in doc("reviews.xml")// entry / title
      satisfies $t1 eq $t2
return $t1
```

After having normalized and translated this query into our algebra, we arrive at the following expression:

$$\Pi_{t1}(\sigma_{\exists t \in (e_2):t1=t2}(e_1))$$

with

$$\begin{aligned} e_1 &:= \Upsilon_{t1:doc("bib.xml")//book/title}(\Box) \\ e_2 &:= \Upsilon_{t2:doc("reviews.xml")//entry/title}(\Box) \end{aligned}$$

It is not hard to detect the basic pattern for existentially quantified queries in this expression. How do we continue from here? Ideally, we would now hand the algebraic expression to an optimizer that determines an efficient query plan based on a cost model. Cost-based optimizers work on the level of query blocks. In the case of XQuery a block corresponds approximately to a FLWOR expression or a quantified expression. Hence, when we are able to merge nested algebraic expressionsinto larger ones, we increase the search space of the query optimizers. As a consequence the query optimizer will often find much more efficient query execution plans. The choice of the best unnesting equivalence to apply is based on heuristics. This heuristic is presented in the form of a decision tree in the next section.

4.3.2. Optimization Strategy

Figure 4.1 shows the decision tree we use for unnesting existentially quantified expressions. Going down the tree from top to bottom, we reach more and more specific rules, which we formally define in Figure 4.2. At the moment, our heuristic consists of applying the most special rewrite rule possible, as the more special rules tend to improve the performance significantly. (For each rule we enumerate all preconditions that have to be met in order to apply this rule, more details follow in the next section.) Let us have a brief look at the decision tree. First of all, we check for an expression $\sigma_{\exists x \in e_2:p}(e_1)$ if e_2 can be evaluated independently of e_1. If not, we leave the expression as it is or evaluate it via an efficiently

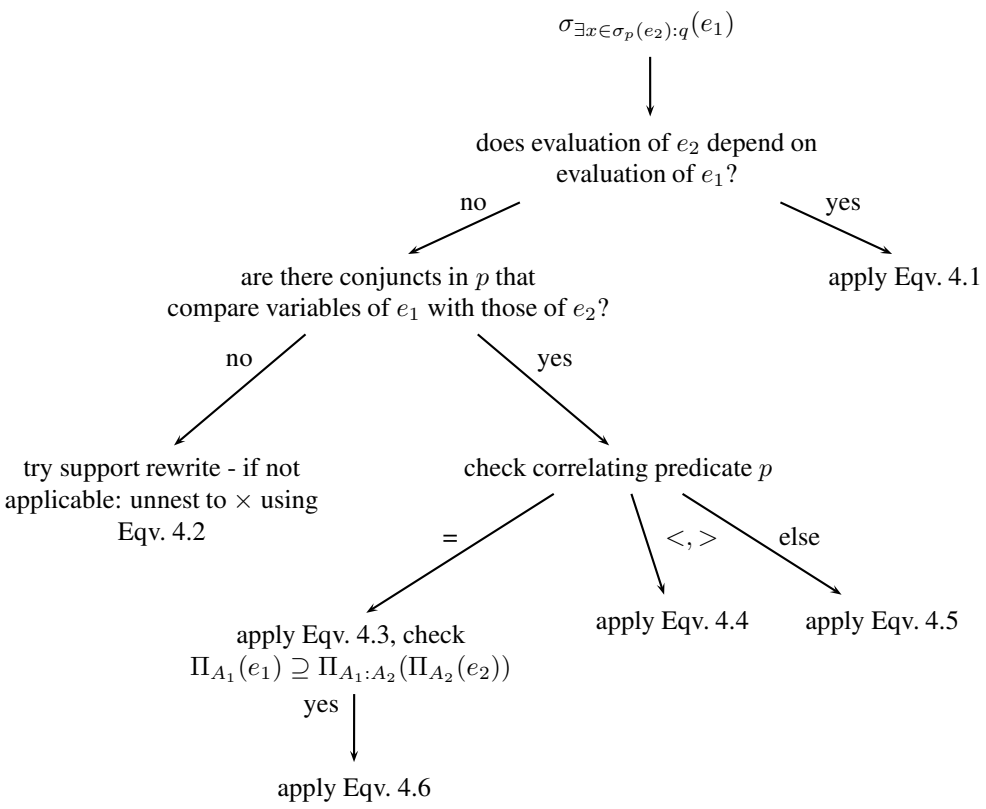


Figure 4.1.: Decision tree for existentially quantified queries

implemented unnest map operator (using Eqv. 4.1) [Gra03, BKHM05]. If yes, we examine the predicate p. If we are not able to correlate the expressions e_1 and e_2 via p, then we unnest the expression with the help of a Cartesian product (using Eqv. 4.2). If p correlates e_1 and e_2, we use different variants of semijoins or grouping/aggregation to unnest the expression (Eqvs. 4.3, 4.4, 4.5, 4.6).

For our motivating example this means that we end up at Eqv. 4.3 (Eqv. 4.6 is not applicable, as `bib.xml` and `reviews.xml` may not contain the same books). Applying Eqv. 4.3 to our example yields

$$\Pi_{t1}(e_1 \ltimes_{t1=t2} e_2)$$

4.3.3. Equivalences for Unnesting

After having outlined the general strategy, we now present the concrete equivalences (see Figure 4.2) and list all prerequisites necessary for applying them.

Equivalence 4.1

Preconditions e_1 and e_2 cannot be evaluated independently (formally speaking, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$).

Basic idea Combine all tuples in e_1 with all tuples in $e_2(e_1)$ via an unnest map operator and then apply p. We need the `tids` to eliminate duplicates.

Equivalence 4.2

Preconditions e_1 and e_2 can be evaluated independently ($\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$).

$$\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(tid_{i_1}(e_1)))) \quad (4.1)$$

$$\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(tid_{i_1}(e_1) \times e_2)) \quad (4.2)$$

$$\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1=A_2 \wedge p} e_2 \quad (4.3)$$

$$\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(e_1) \quad (4.4)$$

$$\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1 \theta A_2 \wedge p} e_2)) \quad (4.5)$$

$$\Pi^D(e_1) \ltimes_{A_1=A_2} (\sigma_p(e_2)) = \sigma_{c>0}(\Pi_{A_1:A_2}(\Gamma_{c;=A_2;count \circ \sigma_p}(e_2))) \quad (4.6)$$

Figure 4.2.: Unnesting equivalences for existentially quantified queries

Basic idea Combine all tuples in e_1 with all tuples in e_2 via a Cartesian product and then apply p. We need the `tids` to eliminate duplicates. This equivalence has to be used if e_1 and e_2 are not correlated via the predicate p. If e_1 and e_2 are correlated, it should only be used if the other equivalences are not applicable.

Equivalence 4.3

Preconditions e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with an equality predicate.

Basic idea Use a semijoin to evaluate the expression. We expect the evaluation of a semijoin operator to be much more efficient than that of a cross product or the nested version of the expression.

Equivalence 4.6

Preconditions Eqv. 4.6 is a special case of Eqv. 4.3. In addition to the preconditions of Eqv. 4.3, $\Pi_{A_1}(e_1) \supseteq \Pi_{A_1:A_2}(\Pi_{A_2}(e_2))$ must hold. This is the case e.g. if both expressions, e_1 and e_2, evaluate the same path expression on the same document.

Basic idea We do not have to redundantly evaluate both e_1 and e_2. It is sufficient to just group all tuples in e_2 on attribute A_2, filter the tuples with predicate p and count the remaining tuples. In the equivalence we denote this operation by a function composition. For existential quantification the number of tuples satisfying p for a certain value A_2 has to be greater than 0.

Equivalence 4.4

Preconditions Same as for Eqv. 4.3, except that e_1 and e_2 are correlated with an inequality predicate. The following table gives the correct assignments for $\theta, \neg\theta$ and $aggr$:

θ	$aggr$
$>, \geq$	min
$<, \leq$	max

Basic idea If we have an inequality comparison operator ($\theta \in \{<, \leq, \geq, >\}$), we just need to compare the value of A_1 to the minimal or maximal value of A_2. For existential quantification a tuple of e_1 satisfies the query predicate if A_1 lies in the range $[\min_{A_2}(e_2), \infty)$ or in the range $(-\infty, \max_{A_2}(e_2)]$, respectively. The resulting nested expression can be unnested with equivalences that are introduced in Sec. 4.5.

We have to be careful when handling the special case $e_2 = \epsilon$. In this case, the predicate $A_1 \theta aggr$ is evaluated to false. Additionally, we must take care

of the semantics of XQuery: In XQuery the sequence of items that functions min or max get as arguments convert these items to $xs : double$. In contrast, the general comparison does not perform such an implicit type conversion, i.e. $xs : string$ is used for the items. For strings we rely on the collation to order the strings and to compute the minimum or maximum.

Equivalence 4.5

Preconditions Same as for Eqv. 4.3, except that e_1 and e_2 are correlated with an arbitrary predicate. This is the most general case for correlated expressions.

Basic idea The general predicate is delegated to a θ-join operator. This has the advantage that the θ-join operator does not need to preserve order (this is done by the semijoin). Non-order-preserving operators can usually be implemented more efficiently. [1]

4.3.4. Support Rewrites

The equivalences for unnesting from the previous section may not be immediately applicable, but with the help of some further rewrite rules, we can bring the expression that is to be optimized into the right form.

For example, take the following expression, in which the tuples bound to variables s and t contain x resp. y as bound variables

$$\exists s \in e_1 : \exists t \in e_2 : x\theta y.$$

None of the equivalences presented in Section 4.3.3 can be applied to this expression directly. However, if we rewrite it to

$$\exists s \in \sigma_{\exists t \in e_2 : x\theta y}(e_1) : \text{true}$$

we can apply equivalence 4.3 and replace the selection with a semijoin:

$$\exists s \in (e_1 \ltimes_{x\theta y} e_2) : \text{true}.$$

When rewriting expressions, we follow two general heuristics. First, we try to reduce the number of free variables in the subexpression we want to unnest. This is mainly achieved by splitting and moving predicates [Ste95]. As all free variables in a subexpression are bound by the enclosing expression, by moving these free variables we try to decouple the subexpression from the enclosing expression as much as possible. The second heuristic involves minimizing the distance between query blocks that are correlated via predicates. These two strategies simplify the unnesting of subexpressions considerably.

In contrast to the unnesting equivalences, which are almost always applied from left to right, the support rewrite rules are usually used in both directions. Hence, we check that we have not applied the rewrite to the same expression before to avoid getting stuck in infinite loops.

Let us now have a look at the rewrite rules (all rules are summarized in Figure 4.3). This list is in no way exhaustive (we just included rules that are needed in the remainder of this paper) and many of the rules are common knowledge and have already been described elsewhere [Bry89, JK84, Ste95].

Equivalence 4.7

Preconditions e_1 and e_2 can be evaluated independently of each other. Furthermore, Eqv. 4.7 may not have been applied to the same subexpression before.

[1] Note that we cannot use the θ-semijoin proposed by [SHP+96] because it is restricted to an unordered context.

$$\exists x \in e_1 : \exists y \in e_2 : p = \exists y \in e_2 : \exists x \in e_1 : p \quad (4.7)$$

$$\exists x \in e_1 : p \wedge q = \exists x \in \sigma_p(e_1) : q \quad (4.8)$$

$$\exists x \in \Pi_A(e_1) : p = \exists x \in e_1 : p \quad (4.9)$$

$$p \wedge \exists x \in e_1 : q = \exists x \in e_1 : p \wedge q \quad (4.10)$$

$$p \vee \exists x \in e_1 : q = \exists x \in e_1 : p \vee q \quad (4.11)$$

$$\sigma_{\exists x \in e_2 : p \wedge \exists y \in e_3 : q}(e_1) = \sigma_{\exists x \in e_2 : p}(\sigma_{\exists y \in e_3 : q}(e_1)) \quad (4.12)$$
$$= \sigma_{\exists y \in e_3 : q}(\sigma_{\exists x \in e_2 : p}(e_1))$$

Figure 4.3.: Support rewrites for existentially quantified queries

Basic idea Since we match fixed query patterns, a wrong order of quantifiers can sometimes prevent the application of an unnesting or support rewrite equivalence. Exchanging quantifiers may be able to solve this problem.

Equivalence 4.8

Preconditions Eqv. 4.8 has not been applied to the same subexpression before.

Basic idea Applied from left to right, it corresponds to the standard *predicate push down* on algebraic expressions – in the opposite direction, it models a *predicate pull up*.[2] This equivalence was used for the motivating example (with $p = \exists y \in e_2 : x \theta y$ and q = true).

Equivalence 4.9

Preconditions The projection may not remove attributes that are needed for the evaluation of the predicate p. Also, Eqv. 4.9 has not been applied to the same subexpression before.

Basic idea Removing or inserting projections in existential quantifiers does not change their result. The rewrite prepares other support rewrites, e.g. predicate pull up, which otherwise would not be applicable for a lack of visibility of bound attributes.

Equivalences 4.10 and 4.11

Preconditions The equivalences have not been applied to the same subexpression before. Also, the free variables in p are not bound by e_1 ($\mathcal{F}(p) \cap \mathcal{A}(e_1) = \emptyset$).

Basic idea We want to move predicates that do not depend on attributes of the subexpression out of that expression. We can do this if the predicate is totally independent of all variables in the subexpression.

Equivalence 4.12

Preconditions Eqv. 4.12 has not been applied to the same subexpression before. Also, all free variables in p and q are bound by the expressions e_1, e_2, or e_3 ($\mathcal{F}(p) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$ and $\mathcal{F}(q) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_3)$).

Basic idea As a selection is order-preserving and commutative, it does not matter in which order the selections are evaluated.

[2] Both rewrites are also called *range nesting* and *(de-)scoping* [Bry89, JK84, Ste95].

4.3.5. Example Queries

We now present more detailed example queries showing the unnesting and support rewrite rules in action. These examples also include measurements on the evaluation times of the different query plans.

Simple Existential Quantification

First, we discuss a variant of the query used in the motivation and in Section 4.2. Optimizing existentially quantified queries in XQuery is motivated by their their frequent usage in general comparisons. General comparisons are implicitly existentially quantified.

```
for  $t1  in  doc("bib.xml")//book/ title
where $t1 = doc("reviews.xml")// entry / title
return  $t1
```

This query considers each title of a book. In the **where** clause, this book title is compared to all titles mentioned in entries of reviews. A book title is returned in the query result if at least one review exists with the same title. It does not matter how many such reviews exist for the same book title. The general comparison returns true when any match is found.

All these steps are made explicit during normalization: The general comparison is turned into a quantified expression that compares the items in both arguments of the general comparison. The comparison is done on atomic values, i.e. for nodes, for example, the typed value is extracted. Then, a value comparison is performed that exploits available type information. In absence of specific type information, a string comparison is performed. As a result of normalization, we replace the general comparison by an existentially quantified expression in the **where** clause.

```
for  $t1  in  doc("bib.xml")//book/ title
where some $t2 in  doc("reviews.xml")// entry / title
      satisfies  $t1  eq  $t2
return $t1
```

We translate the normalized query into

$$\Pi_{t1}(\sigma_{\exists t3 \in e_2 : t1=t2}(e_1))$$

where

$$e_1 := \Upsilon_{t1:doc1//book/title}(\Box)$$
$$e_2 := \Upsilon_{t2:doc2//entry/title}(\Box)$$

and

```
doc1 := doc("bib.xml")
doc2 := doc("reviews.xml")
```

Unnesting The translated query almost directly matches the conditions for Eqv. 4.3. We only need to push the range predicate of the nested query into the range expression using support rewrite 4.8. We will use this rewrite quite frequently in our examples. As result of these steps we get

$$\begin{aligned} & \Pi_{t1}(\sigma_{\exists t3 \in e_2 : t1=t2}(e_1)) \\ \overset{(4.8)}{=} \; & \Pi_{t1}(\sigma_{\exists t3 \in \sigma_{t1=t2}(e_2) : \mathrm{true}}(e_1)) \\ \overset{(4.3)}{=} \; & \Pi_{t1}(e_1 \ltimes_{t1=t2} e_2). \end{aligned}$$

Evaluation Before we discuss the experimental results, let us briefly describe the experimental setup. All queries were implemented and evaluated in our native XML database system Natix. They were executed with warm buffer on documents that fit into the database buffer. We only report elapsed times because query execution was CPU-bound.

The data sets we used are based on the XQuery Use Cases "XMP" and "R". "XMP" contains data on books, authors, editors, reviews and so on, while "R" describes an auction site with users, items, bids, etc. As in this first example query, we will sometimes use the fact that child nodes occur exactly once below their parents. In the appendix A.3 we give further details of the experimental setup.

The performance of these two evaluation plans is compared in the following table.

Size	100	1000	10000
Nested	0.10 s	1.83 s	175.80 s
Unnested	0.08 s	0.09 s	0.20 s

The measurements clearly show that the unnested query plan scales better than the nested plan. When the document size of the input increases from 1000 to 10000 the execution time of the nested query increases by a factor of 1000 – a direct consequence of the nested-loop-evaluation. On the other hand, for the efficient hash-based implementation of the semijoin in the unnested query the execution time only doubles when both the input size of both input documents increases by a factor of 100.

Existential Quantification vs. Grouping

Existential Quantification Using `exists` Existential quantification might be expressed in different ways. Instead of using a quantified expression, it is also possible to use the function `empty` or check if counting evaluates to zero. The following example illustrates a third alternative using the function `exists`. This query returns all the books having at least on author containing the string "Suciu" in its name.

```
let $d1 := doc("bib.xml")
for $b1 in $d1//book
where exists (for $b2 := $d1//book,
                  $a2 in $b2/author
              where contains($a2, "Suciu")
                and $b1 is $b2
              return $b2)
return $b1
```

During normalization we extract the complex FLWR expression in the argument of function `exists` into a new **let** clause.

```
let $d1 := doc("bib.xml")
for $b1 in $d1//book
let $b3 := (for $b2 in $d1//book,
                $a2 in $b2/author
            where contains($a2, "Suciu")
              and $b1 is $b2
            return $b2)
where exists ($b3)
return $b1
```

Then, the translation of the query yields:

$$\Pi_{b1}(\sigma_{fn::exists(b3)}(\chi_{b3:e_3}(e_1))).$$

where

$$\begin{aligned} e_1 &:= \Upsilon_{b1:d1//book}(\chi_{d1:doc1}(\Box)) \\ e_2 &:= \Upsilon_{a2:b2/author}(\Upsilon_{b2:d1//book}(\Box)) \\ e_3 &:= \Pi_{b2}(\sigma_{b1=b2 \wedge contains(a2,"Suciu")}(e_2)) \end{aligned}$$

and

```
doc1 := doc("bib.xml")
```

Unnesting by Detecting Grouping Since e_1 and e_3 differ only in the retrieval and filter on the authors name, the expression can be unnested by using Eqv. 4.29 or 4.27 — see Section 4.5 for details. Note, that the condition $e_1 = \Pi_{A_1:A_2}(\Pi^D_{A_2}(e_2))$ holds and that we can apply the filter operation after associating tuples to groups. Thus, we can even apply Eqv. 4.30.

$$\begin{array}{ll} & \Pi_{a1}(\sigma_{fn::exists(b3)}(\chi_{b3:\Pi_{b2}(\sigma_{b1=b2\wedge contains(a2,"Suciu")}(e_2))}(e_1))) \\ \overset{(4.30)}{=} & \Pi_{a1}(\sigma_{fn::exists(b3)}(\Pi_{a1:a2}(\Gamma_{b3;=b2;id\circ\sigma_{contains(a2,"Suciu")}}(e_2)))) \end{array}$$

This is not really efficient, because we materialize all tuples that belong to a group only to check if the group was not empty afterwards. Thus, it is better to replace function `exists` by the expression `0 > count` before unnesting. This corresponds to the second alternative of expressing existential quantification mentioned above. Then, we get the following sequence of rewrites:

$$\begin{array}{ll} & \Pi_{b1}(\sigma_{fn::exists(b3)}(\chi_{b3:\Pi_{b2}(\sigma_{b1=b2\wedge contains(a2,"Suciu")}(e_2)}(e_1))) \\ = & \Pi_{b1}(\sigma_{count(b3)>0}(\chi_{b3:\Pi_{b2}(\sigma_{b1=b2\wedge contains(a2,"Suciu")}(e_2)}(e_1)))) \\ = & \Pi_{b1}(\sigma_{c>0}(\chi_{c:count(\Pi_{b2}(\sigma_{b1=b2\wedge contains(a2,"Suciu")}(e_2)}(e_1))))) \\ \overset{(4.30)}{=} & \Pi_{b1}(\sigma_{c>0}(\Gamma_{b3;=b2;count\circ\sigma_{contains(a2,"Suciu")}}(e_2)) \end{array}$$

Existential Quantification Using Quantified Expression When we formulate this query using a quantified expression we get the following query:

```
let  $d1 := doc("bib.xml")
for  $b1 in $d1//book
where some $b2 in $d1//book,
           $a2 in $b2/author
      satisfies  contains ($a2, "Suciu") and $b1 is $b2
return $b1
```

During normalization the query remains unchanged, and the translation results in

$$\Pi_{b1}(\sigma_{\exists b\in e_2:e_3}(e_1)).$$

where

$$\begin{array}{lcl} e_1 & := & \Upsilon_{b1:d1//book}(\chi_{d1:doc1}(\Box)) \\ e_2 & := & \Upsilon_{a2:b2/author}(\Upsilon_{b2:d1//book}(\Box)) \\ e_3 & := & b1 = b2 \wedge contains(a2, "Suciu") \end{array}$$

and

```
doc1 := doc("bib.xml")
```

Unnesting Quantified Expression Again, we have two choices to unnest this query. In a first attempt start unnesting by pushing the range predicate of the existential quantifier into the range expression using Eqv. 4.8. Then we push this selection down into expression e_3. In the last unnesting step we apply Eqv. 4.3.

$$\begin{array}{ll} & \Pi_{b1}(\sigma_{\exists b3\in e_2:e_3}(e_1)) \\ \overset{(4.8)}{=} & \Pi_{b1}(\sigma_{\exists b3\in\sigma_{e_3}(e_2):\mathrm{true}}(e_1)) \\ \overset{(4.3)}{=} & \Pi_{b1}(e_1 \ltimes_{b1=b2} (\sigma_{contains(a2,''Suciu'')}(e_2))) \end{array}$$

As already discussed before for this query, we may also exploit the fact that the condition $\Pi_{A_1}(e_1) \supseteq \Pi_{A_1:A_2}(\Pi_{A_2}(e_2))$ holds. This observation allows us to apply Eqv. 4.6 after applying Eqv. 4.8.

$$\begin{aligned} & \Pi_{b1}(\sigma_{\exists b3 \in e_2 : e_3}(e_1)) \\ \overset{(4.8)}{=} \quad & \Pi_{b1}(\sigma_{\exists b3 \in \sigma_{e_3} e_2 : \mathrm{true}}(e_1)) \\ \overset{(4.6)}{=} \quad & \Pi_{b1}(\sigma_{c>0}(\Gamma_{c;=b2;count \circ contains(a2,''Suciu'')}(e_2))) \end{aligned}$$

An encouraging result of the discussion on this query is that different formulations of the same query lead us to the same unnested expression. This is a good indication for the general applicability of our set of rewrites and unnesting equivalences.

Evaluation In the table below, we summarize the execution times for the three presented expressions. The tremendous effect of unnesting can also be seen in this case. In addition, we observe a performance gain in the third evaluation plan, which is caused by avoiding one scan of the input document.

Size	100	1000	10000
Nested	0.04 s	1.31 s	138.8 s
Semijoin	0.03 s	0.05 s	0.30 s
Grouping	0.02 s	0.02 s	0.02 s

Exchanging Quantifiers

With the following example query, we show how an expression can be rewritten using Eqv. 4.7 to allow for more efficient unnesting techniques. In the query below we want to determine all users of an auction site who are actively bidding on at least one item:

```
for $u in doc("users.xml")// usertuple
where some $i in doc("items.xml")// itemtuple
        satisfies some $b in doc("bids.xml")// bidtuple
                  satisfies ($u/userid eq $b/userid and
                             $i/itemno eq $b/itemno)
return $u/name
```

Following the normalization steps introduced in Section 3.4, we move the path expressions in the innermost range predicate into new **let** clauses in the quantified subexpressions.

```
for $u in doc("users.xml")// usertuple
let $un := $u/name
let $uu := $u/userid
where some $i in in doc("items.xml")// itemtuple
       let $ii := $i/itemno
        satisfies some $b in doc("bids.xml")// bidtuple
                  let $bu := $b/userid
                  let $bi := $b/itemno
                   satisfies ($uu eq $bu and $ii eq $bi)
return $un
```

Translating the above into our algebra results in the following expression:

$$\Pi_{un}(\chi_{un:u/name}(\sigma_{\exists it \in (e_2) : \exists bt \in (e_3) : e_4}(e_1)))$$

where

$$
\begin{aligned}
e_1 &:= \chi_{uu:u/userid}(\Upsilon_{u:doc1//usertuple}(\Box)) \\
e_2 &:= \chi_{ii:i/itemno}(\Upsilon_{i:doc2//itemtuple}(\Box)) \\
e_3 &:= \chi_{bi:b/itemno}(\chi_{bu:b/userid}(\\
&\qquad \Upsilon_{b:doc3//bidtuple}(\Box))) \\
e_4 &= uu = bu \wedge ii = bi
\end{aligned}
$$

and

```
doc1 := doc("users.xml")
doc2 := doc("items.xml")
doc3 := doc("bids.xml")
```

Note that during the translation we exploit the fact that the child nodes of `itemtuple`, `bidtuple`, and `usertuple` occur exactly once. Since predicate e_4 references variables bound in e_1, e_2, and e_3, none of the more efficient unnesting equivalences on the lower right hand side of the decision tree are applicable immediately. However, using some of the support rewrite rules, we can remedy this situation. First, we are going to present a naive approach to unnesting the above algebraic expression. Then, we will show how to optimize it in a more clever way.

Naive Unnesting As e_1 and e_2 can be evaluated independently of each other and they are not correlated in any way, we can apply Eqv. 4.2. After having pushed down the predicate e_4 (see Eqv. 4.8), we can apply Eqv. 4.3 connecting e_3 via a semijoin:

$$
\begin{aligned}
& \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists it\in(e_2):\exists bt\in(e_3):e_4}(e_1))) \\
\overset{(4.2)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt\in(e_3):e_4}(tid_{p_1}(e_1)\times e_2)))) \\
\overset{(4.8)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt\in(\sigma_{e_4}(e_3)):true}(tid_{p_1}(e_1)\times e_2)))) \\
\overset{(4.3)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}((tid_{p_1}(e_1)\times e_2)\ltimes_{e_4} e_3)))
\end{aligned}
$$

Improved Unnesting However, we can do better than that and avoid using the Cartesian product. If we first reorder the quantifiers $\exists it \in (e_2) : \exists bt \in (e_3) : e_4$ using Eqv. 4.7 and then push down the first part of the predicate e_4, we can apply Eqv. 4.3. After having pushed down the second part of e_4, we can apply Eqv. 4.3 again, arriving at an expression containing two semijoins:

$$
\begin{aligned}
& \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists it\in(e_2):\exists bt\in(e_3):e_4}(e_1))) \\
\overset{(4.7)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists bt\in(e_3):\exists it\in(e_2):e_4}(e_1))) \\
\overset{(4.8)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists bt\in(e_3):\exists it\in(\sigma_{ii=bi}(e_2)):uu=bu}(e_1))) \\
\overset{(4.8)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists bt\in\sigma_{\exists it\in(\sigma_{ii=bi}(e_2))}(e_3):uu=bu}(e_1))) \\
\overset{(4.3)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists bt\in(e_3\ltimes_{ii=bi}e_2):uu=bu}(e_1))) \\
\overset{(4.8)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists bt\in(\sigma_{uu=bu}(e_3\ltimes_{ii=bi}e_2)):true}(e_1))) \\
\overset{(4.3)}{=}\ & \Pi_{un}(\chi_{un:u/name}(e_1\ltimes_{uu=bu}(e_3\ltimes_{ii=bi} e_2)))
\end{aligned}
$$

Evaluation Running the nested, the naively unnested, and the improved unnested version, we acquired the following averaged running times (in seconds).

Size	100	1000	10000
Nested	10.42s	3944.71s	∞
Naively Unnested	0.16s	8.45s	860.69s
Improved Unnested	0.08s	0.12s	0.56s

The nested version is clearly the slowest variant (for a document size of 10000 nodes we aborted the execution after three hours). While the naively unnested version already improves the performance by several orders of magnitude, we can decrease the evaluation time even further below one second for the largest document size by eliminating the Cartesian product.

Non-Equality Correlation Predicates

Correlation predicates which are no equality predicates are more difficult to evaluate. While predicates involving equality can be mapped to equijoins, we now have to employ more general θ-joins (which usually are more expensive to evaluate).

Our example query illustrates a simple integrity check testing if there are any bids which are bound to fail, as the reserve price of an item has not been met:

```
for $b in doc("bids.xml")// bidtuple
where some $i in doc("items.xml")// itemtuple
                                    [itemno eq $b/itemno]
      satisfies $i/reserveprice gt $b/bid
return
  <failcheck>
    { $b/itemno, $b/userid }
  </failcheck>
```

The normalized query introduces several **let** clauses and moves complex expressions out of the **where** and **return** clause.

```
for $b in doc("bids.xml")// bidtuple
let $bn := $b/itemno
let $bb := $b/bid
let $bu := $b/userid
let $bs := ($bn, $bu)
let $res := <failcheck> { $bs } </failcheck>
where some $i in doc("items.xml")// itemtuple
       let $in := $i/itemno
       let $ir := $i/reserveprice
      where $in eq $bn
       satisfies $ir gt $bb
return $res
```

During translation we exploit the fact that each child element of `bidtuple` and `itemtuple` occurs only once (we know this from the DTD). As a result of translating the normalized query into our algebra, we get:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,bs)}(\chi_{bs:(bn,bu)}(\chi_{bu:b/userid}(\sigma_{\exists it \in \sigma_{in=bn}(e_2):ir>bb}(e_1)))))$$

where

$$e_1 := \chi_{bn:b/itemno}(\chi_{bb:b/bid}(\Upsilon_{b:doc1//bidtuple}(\Box)))$$

$$e_2 := \chi_{ir:i/reserveprice}(\chi_{in:i/itemno}(\Upsilon_{i:doc2//itemtuple}(\Box)))$$

and

```
doc1 := doc("bids.xml")
doc2 := doc("items.xml")
s1   := "failcheck"
```

Unnesting To establish the pattern for quantified queries we push down the predicate $ir > bb$ using Eqv. 4.8. Then, we can directly apply Eqv. 4.5 for unnesting general non-equality correlating predicates:

$$
\begin{aligned}
&= \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,bs)}(\chi_{bs:(bn,bu)}(\chi_{bu:b/userid}(\sigma_{\exists it \in \sigma_{in=bn}(e_2):ir>bb}(e_1)))))\\
&\overset{(4.8)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,bs)}(\chi_{bs:(bn,bu)}(\chi_{bu:b/userid}(\sigma_{\exists it \in \sigma_{in=bn \wedge ir>bb}(e_2):\text{true}}(e_1)))))\\
&\overset{(4.5)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,bs)}(\chi_{bs:(bn,bu)}(\chi_{bu:b/userid}(e_1 \ltimes_{bn=bn' \wedge bb=bb'} (\\
&\quad \Pi_{bn':bn,bb':bb}(e_1 \bowtie_{in=bn \wedge ir>bb} e_2))))))
\end{aligned}
$$

At first glance this may look less efficient than the original algebraic expression, but as we can evaluate the θ-join via a more efficient block-wise nested-loop join, rather than using a naive nested-loop join, we can gain performance here. Since ordering is not important in the range expression of the existential quantifier, we are given more leeway in tuning the parameters of the semijoin and θ-join (e.g. order of join arguments, hash table size). In this particular case we could also go a step further and rewrite $e_1 \bowtie_{in=bn \wedge ir>bb} e_2$ into $\sigma_{ir>bb}(e_1 \bowtie_{in=bn} e_2)$ and use an even more efficient hash-join algorithm to evaluate the join between e_1 and e_2. [3]

Evaluation The following table summarizes our experimental results for this query:

Size	100	1000	10000
Nested	0.16s	5.21s	451.97s
Unnested	0.10s	0.25s	10.63s

Even though we duplicate a subexpression, e_1, we observe significant gains in efficiency when unnesting. The improvements are mainly due to employing efficient join implementations that become an option after identifying subexpressions insensitive to document order.

Complex Correlation

In the following example query, we demonstrate how complex correlation predicates between query blocks can be untangled. We retrieve all users who bid on an item (which they do not offer themselves) and where the bid is at least twice as high as the reserve price:

```
for $u in doc("users.xml")// usertuple
where some $i in doc("items.xml")// itemtuple
      satisfies ($i/offeredby ne $u/userid
            and some $b in doc("bids.xml")// bidtuple
                  satisfies ($b/userid eq $u/userid
                        and $b/itemno eq $i/itemno
                        and ($b/bid cast as xs:double) gt
                              (2.0 * $i/reserveprice )))
return $u/userid
```

Normalizing and translating the XQuery expression into our algebra, we get:

```
for $u in doc("users.xml")// usertuple
let $ui := $u/userid
where some $i in doc("items.xml")// itemtuple
      let $io := $i/offeredby
      let $ir := $i/reserveprice
      let $in := $i/itemno
      satisfies ($io ne $ui and
                  some $b in doc("bids.xml")// bidtuple
```

[3] We refrained from doing so, as we wanted to measure the performance gains possible for unnesting a general correlation predicate.

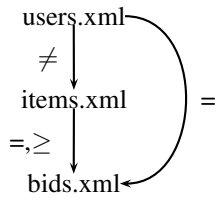


Figure 4.4.: Dependencies

```
let $bi := $b/userid
let $bb := $b/bid cast as xs:double
let $bn := $b/itemno
satisfies ($bi eq $ui and
           $bu eq $in and
           $bb gt 2.0 * $ir ))
return $ui
```

Translating this expression into our algebra yields:

$$\Pi_{ui}(\sigma_{\exists it \in e_2:(io \neq ui \wedge \exists bt \in e_3:e_4)}(e_1))$$

where

$$e_1 := \chi_{ui:u/userid}(\Upsilon_{u:doc1//usertuple}(\Box))$$
$$e_2 := \chi_{in:i/itemno}(\chi_{ir:i/reserveprice}(\chi_{io:i/offeredby}(\Upsilon_{i:doc2//itemtuple}(\Box))))$$
$$e_3 := \chi_{bn:b/itemno}(\chi_{bb:b/bid}(\chi_{bi:b/userid}(\Upsilon_{b:doc3//bidtuple}(\Box))))$$
$$e_4 := bi = ui \wedge bn = in \wedge bb > 2.0 \cdot ir$$

and

```
doc1 := doc("users.xml")
doc2 := doc("items.xml")
doc3 := doc("bids.xml")
```

Although the correlation predicate looks quite complicated, our unnesting techniques are powerful enough to handle even this case. The graph in Fig. 4.4 depicts the complexity of the correlation predicate by showing how the query blocks accessing the different documents (represented as nodes) are connected via the predicates (represented as edges). The edge runs from the query block that binds a variable to the nested query block that uses this binding:

We present two different ways to unnest the above algebraic expression. One involves a direct unnesting via a semijoin, the other an indirect unnesting via a Cartesian product (which is eliminated later on).

Semijoin 1 This approach is quite straightforward, as we apply Eqv. 4.5, pull a part of the join predicate into a selection outside the join, and then apply Eqv. 4.5 again in order to unnest the doubly nested expression:

$$\begin{aligned}
& \Pi_{ui}(\sigma_{\exists it \in e_2:(io \neq ui \wedge \exists bt \in e_3:e_4)}(e_1)) \\
\overset{(4.5)}{=} \; & \Pi_{ui}(e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'} (\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}(e_1 \bowtie_{io \neq ui \wedge \exists bt \in e_3:e_4} e_2))) \\
= \; & \Pi_{ui}(e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'} (\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}(\sigma_{\exists bt \in e_3:e_4}(e_1 \bowtie_{io \neq ui} e_2)))) \\
\overset{(4.5)}{=} \; & \Pi_{ui}(e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'} (\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}((e_1 \bowtie_{io \neq ui} e_2) \ltimes_{\mathcal{A}(e_1,e_2)=\mathcal{A}(e_1,e_2)'} \\
& \quad (\Pi_{\mathcal{A}(e_1,e_2)':\mathcal{A}(e_1,e_2)}((e_1 \bowtie_{io \neq ui} e_2) \bowtie_{e_4} e_3)))))
\end{aligned}$$

Semijoin 2 Although we advised against using Cartesian products, we can use Eqv.4.2 in a first step, then pull in part of the selection predicate into the Cartesian product to change

it into a join, and finally apply Eqv. 4.5, introducing a semijoin:

$$
\begin{aligned}
& \Pi_{ui}(\sigma_{\exists it \in e_2:(io \neq ui \wedge \exists bt \in e_3:e_4)}(e_1)) \\
\overset{(4.2)}{=}\ & \Pi_{ui}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}((\sigma_{io \neq ui \wedge \exists bt \in (e_3):e_4}(tid_{p_1}(e_1) \times e_2)))) \\
\overset{(4.12)}{=}\ & \Pi_{ui}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt \in (e_3):e_4}(\sigma_{io \neq ui}(tid_{p_1}(e_1) \times e_2)))) \\
\overset{(4.5)}{=}\ & \Pi_{ui}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}(\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}((tid_{p_1}(e_1) \bowtie_{io \neq ui} e_2) \ltimes_{\mathcal{A}(e_1,e_2)=\mathcal{A}(e_1,e_2)'} \\
& \quad (\Pi_{\mathcal{A}(e_1,e_2)':\mathcal{A}(e_1,e_2)}((tid_{p_1}(e_1) \bowtie_{io \neq ui} e_2) \bowtie_{e_4} e_3)))))
\end{aligned}
$$

The main difference between this expression and the first semijoin variant is the fact that in the first variant, all θ-joins need *not* be order-preserving (the semijoin with e_1 determines the final order), while here the first θ-join between e_1 and e_2 needs to be order-preserving. In both variants we can optimize the expression $(e_1 \bowtie_{io \neq ui} e_2) \bowtie_{e_4} e_3$, further using standard join ordering techniques (in this way, we get two joins involving equality predicates):

$$(e_1 \bowtie_{io \neq ui} e_2) \bowtie_{e_4} e_3 \quad = \quad (e_3 \bowtie_{bi = ui} e_1) \bowtie_{bn = in \wedge io \neq ui \wedge bb > 2.0 \cdot ir} e_2$$

Evaluation The following table shows the results from our measurements. As can be seen clearly, both unnested variants outperform the nested version easily. Again, Semijoin 2 is slower because we require the first θ-join to be order-preserving while for Semijoin 1 no such restriction exists for any of the θ-joins.

Size	100	1000	10000
Nested	56.69s	3041.22s	∞
Semijoin 1	0.21s	0.80s	81.21s
Semijoin 2	0.63s	14.25s	1176.2s

General Comparisons

In the previous sections, we assumed all comparisons to be value-based. Now we show how we can handle general comparisons with our approach. The main idea is to transform the general comparisons into explicit existentially quantified expressions with value comparisons during normalization. Then, after the translation into the algebra, we use our techniques to unnest these expressions. Following that, we can continue with unnesting the actual nested query as shown before.

Consider the following example query, in which we are looking for books that are sold below the price mentioned in some review (e.g. suggested retail price):

```
for $b in doc("bib.xml")//book
where some $e in doc("reviews.xml")//entry [ title = $b/title ]
      satisfies $e/price > $b/price
return
  <cheap-book>
    { $b/title , $b/price }
  </cheap-book>
```

During normalization we expand the range expressions of the quantified queries to FLWR expressions. Normalization of the quantified queries ensures that all comparisons become value comparisons[4]:

```
for $b in doc("bib.xml")//book
let $bt := $b/title
```

[4]Here and in the sequel we omit conversions on the sequences and types for readability. We refer to [DFF+07] for details.

```
let $bp  := $b/price
let $bs  := ($bt, $bp)
let $res := <cheap−book> { $bs } </cheap−book>
where some $e in doc("reviews.xml")// entry
        let $et := $e/ title
        let $ep := $e/price
       where some $ets in $et
                satisfies some $bts in $bt
                                satisfies $ets eq $bts
        satisfies some $eps in $ep
                        satisfies some $bps in $bp
                                        satisfies $eps gt $bps
return $res
```

Translating this into our algebra yields:

$$\Pi_{res}(\sigma_{e_2}(e_0))$$

where

$$\begin{aligned}
e_0 &:= \chi_{res:\mathcal{C}(\text{elem},s1,bs)}(\chi_{bs:(bt,bp)}(\chi_{bp:b/price}(\\
&\quad \chi_{bt:b/title}(\Upsilon_{b:doc1//book}(\Box)))))\\
e_1 &:= \chi_{ep:e/price}(\chi_{et:e/title}(\\
&\quad \Upsilon_{e:doc2//entry}(\Box)))\\
e_2 &:= \exists et1 \in (\sigma_{e_3}(e_1)) : e_6\\
e_3 &:= \exists et2 \in e_4 : \exists bt1 \in e_5 : ets = bts\\
e_4 &:= \Upsilon_{ets:et}(\Box)\\
e_5 &:= \Upsilon_{bts:bt}(\Box)
\end{aligned}$$

$$\begin{aligned}
e_6 &:= \exists et3 \in e_7 :\\
&\quad \exists bt2 \in e_8 : eps > bps\\
e_7 &:= \Upsilon_{eps:ep}(\Box)\\
e_8 &:= \Upsilon_{bps:bp}(\Box)
\end{aligned}$$

and

```
doc1 := doc("bib.xml")
doc2 := doc("reviews.xml")
s1   := "cheap-book"
```

Dependencies between different expressions (the evaluation of e_5 and e_8 depends on e_0, while that of e_4 and e_7 depends on e_1) do not make our job any easier. That means that in the first step of unnesting the introduced quantified expressions, we are forced to use Eqv. 4.1. However, we can improve our situation by decoupling the range expression in e_2, $\sigma_{e_3}(e_1)$, from the outer query block. We do this by pushing the independent parts of the predicates in e_2 into the range expression and moving the dependent parts into the range predicate:

$$\begin{aligned}
e_2 &= \exists et1 \in (\sigma_{e_3}(e_1)) : e_6\\
&\overset{(4.1)}{=} \exists et1 \in (\Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt1 \in e_5 : ets=bts}(\Upsilon_{\mathcal{A}(e_4):e4}(tid_{i_1}(e_1))))) :\\
&\quad \exists et3 \in e_7 : \exists bt2 \in e_8 : eps > bps\\
&\overset{(4.8)}{=} \exists et1 \in (\sigma_{\exists et3 \in e_7 : \exists bt2 \in e_8 : eps > bps}(\Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\\
&\quad \sigma_{\exists bt1 \in e_5 : ets=bts}(\Upsilon_{\mathcal{A}(e_4):e4}(tid_{i_1}(e_1)))))) : \text{true}\\
&\overset{(4.1)}{=} \exists et1 \in (\Pi^{tid_{i_2}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt2 \in e_8 : eps > bps}(\Upsilon_{\mathcal{A}(e_7):e_7}(tid_{i_2}(\\
&\quad \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt1 \in e_5 : ets=bts}(\Upsilon_{\mathcal{A}(e_4):e4}(tid_{i_1}(e_1))))))))))\\
&\overset{(4.9)}{=} \exists et1 \in (\sigma_{\exists bt2 \in e_8 : eps > bps}(\Upsilon_{\mathcal{A}(e_7):e_7}(\sigma_{\exists bt1 \in e_5 : ets=bts}(\Upsilon_{\mathcal{A}(e_4):e4}(e_1))))))\\
&\overset{(4.8)}{=} \exists et1 \in (\Upsilon_{\mathcal{A}(e_7):e_7}(\Upsilon_{\mathcal{A}(e_4):e4}(e_1))) :\\
&\quad (\exists bt2 \in e_8 : eps > bps) \wedge (\exists bt1 \in e_5 : ets = bts)
\end{aligned}$$

In the last but one step, we also eliminate the tid operators as they are not needed anymore (as both projections on the tids have been removed). To be able to apply Eqv. 4.8 twice in

the last step, we exchanged the positions of $\Upsilon_{\mathcal{A}(e_7):e_7}$ and $\sigma_{\exists bt2 \in e_5 : ets = bts}$, which poses no problem, as e_7 is not connected to the selection predicate in any way.

After having removed the level of nesting introduced by the general comparisons, we could now continue with the unnesting of the actual query. As we have already shown how to proceed with nested queries containing value comparisons in the previous examples, we leave it out here.

4.4. Universal Quantifiers

We start this section with an example to motivate unnesting queries containing universal quantifiers. Then we introduce a general optimization strategy and present rules for unnesting and rewriting algebraic expressions. The application of these rules to typical query classes follows.

4.4.1. Motivating Example

As a motivating example for universal quantifiers we present a query in which we want to find all auction items that only have valid bids (all bids are at least as high as the reserve price):

for $i **in** doc("items.xml")// itemtuple
where every $b **in** doc("bids.xml")// bidtuple
[itemno **eq** $i/itemno]
satisfies $b/bid ge $i/ reserveprice
return $i/itemno

Normalizing and translating this query results in the following algebraic expression:

$$\Pi_{ii}(\sigma_{\forall bt \in \sigma_{bi=ii}(e_2):bb \geq ir}(e_1))$$

where

$$\begin{aligned} e_1 &:= \chi_{ii:i/itemno}(\chi_{ir:i/reserveprice}(\Upsilon_{i:doc(\text{"}items.xml\text{"})//itemtuple}(\Box))) \\ e_2 &:= \chi_{bb:b/bid}(\chi_{bi:b/itemno}(\Upsilon_{b:doc(\text{"}bids.xml\text{"})//bidtuple}(\Box))) \end{aligned}$$

The pattern for universally quantified expressions can be easily identified in the translated version of the query. The general strategy for unnesting these expressions is given in the following section.

4.4.2. Optimization Strategy

The strategy for unnesting universally quantified expressions is very similar to that used for existentially quantified expressions. (See Figure 4.5 for the decision tree and Figure 4.6 for the equivalences). Again, we try to apply the most special rewrite rule possible.

For our motivation example, this means that we end up at Eqv. 4.15. Applying this equivalence to our example yields (note that we have to negate the range predicate in the antijoin):

$$\Pi_{ii}((e_1) \triangleright_{bi=ii \wedge bb<ir} (e_2))$$

Let us give a word of caution related to pushing conjuncts of p that only refer to e_2 (conjuncts pushed into e_1 can be handled as in the case of existential quantification). If a conjunct pushed into e_2 filters out even a single tuple, then the quantified expression returns an empty answer. During query evaluation, this can be used by first evaluating e_2 and aborting the evaluation immediately after a tuple is filtered out by a pushed conjunct of p.

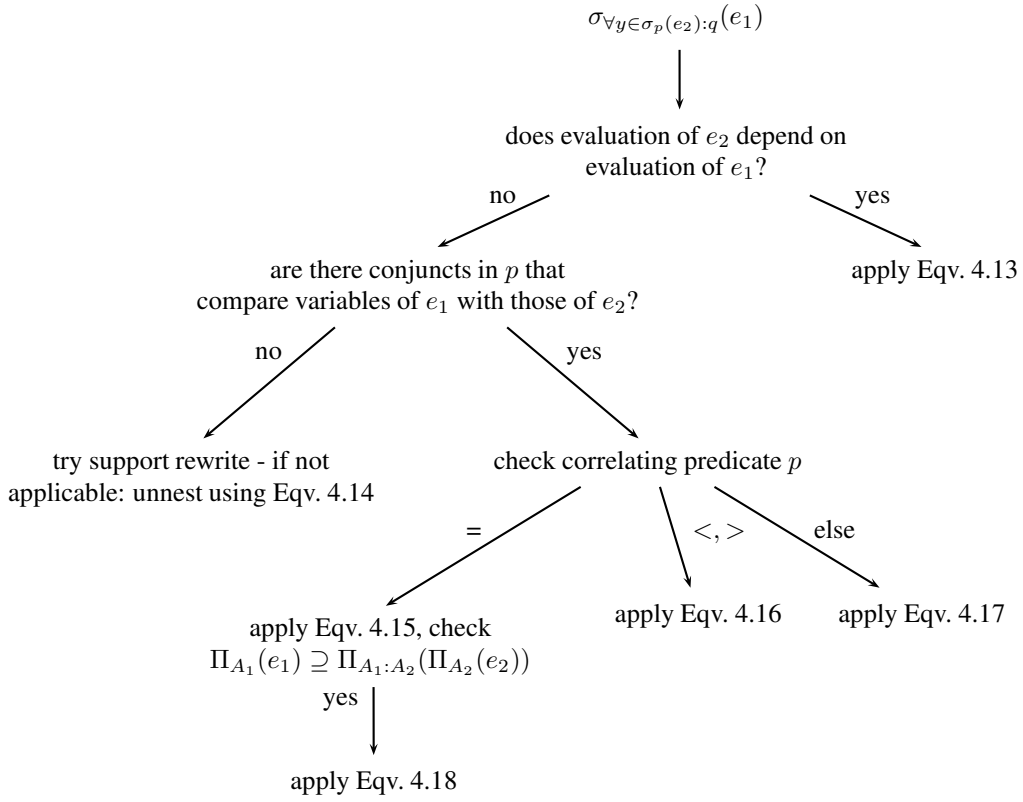


Figure 4.5.: Decision tree for universally quantified queries

4.4.3. Equivalences for Unnesting

Figure 4.6 lists the equivalences for universal quantification. For each unnesting equivalence in Section 4.3, we have a universally quantified counterpart. We proceed by discussing the equivalences in more detail:

$$\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{A_1=A_3} \Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1))) \quad (4.13)$$

$$\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{\neg p} e_2 \quad (4.14)$$

$$\sigma_{\forall x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) = e_1 \triangleright_{A_1=A_2 \wedge \neg p} e_2 \quad (4.15)$$

$$\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \neg \theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(e_1) \quad (4.16)$$

$$\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = (e_1) \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1 \theta A_2 \wedge \neg p} e_2)) \quad (4.17)$$

$$\Pi^D(e_1) \triangleright_{A_1=A_2} (\sigma_p(e_2)) = \sigma_{c=0}(\Pi_{A_1:A_2}(\Gamma_{c;=A_2;count \circ \sigma_p}(e_2))) \quad (4.18)$$

Figure 4.6.: Unnesting equivalences for universally quantified queries

Equivalence 4.13

Preconditions Expression e_1 and e_2 cannot be evaluated independently, i.e. $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$.

Basic idea We use an unnest map operator to evaluate the subexpression e_2 depending on the current tuple in e_1. If we find at least one tuple that satisfies the

negation of the predicate p, then the corresponding tuple in the outer expression e_1 finds a join partner and will be filtered out by the antijoin.

Equivalence 4.14

Preconditions Expression e_1 and e_2 can be evaluated independently, i.e. $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$.

Basic idea At first glance, this equivalence looks quite simple. However, when p does not correlate e_1 and e_2, then the evaluation of this expression has to be done in a nested-loop fashion.

Equivalence 4.15

Preconditions The evaluation of e_2 does not depend on e_1, that is, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$ and e_1 and e_2 are correlated by an equality predicate.

Basic idea We fall back on an antijoin operator. As e_2 does not depend on e_1, we do not need the unnest map found in Eqv. 4.13.

Equivalence 4.18

Preconditions This equivalence is a special case of Eqv. 4.15. An additional precondition is $\Pi_{A_1}(e_1) \supseteq \Pi_{A_1:A_2}(\Pi_{A_2}(e_2))$.

Basic idea This equivalence is the counterpart of Eqv. 4.6 for existential quantification. It avoids to evaluate the same subexpression multiple times if the condition check $\Pi_{A_1}(e_1) \supseteq \Pi_{A_1:A_2}(\Pi_{A_2}(e_2))$ holds. For universal quantification we need to make sure that no tuple exists that satisfies the predicate p.

Equivalence 4.16

Preconditions Same preconditions as for Eqv. 4.15. Depending on the comparison operator θ in p, we have the following assignments:

θ	$\neg\theta$	$aggr$
$>, \geq$	$\leq, <$	min
$<, \leq$	$\geq, >$	max

Basic idea If the comparison operator $\theta \in \{<, \leq, \geq, >\}$, we just need to compare the value of A_1 to the minimal or maximal value of A_2, respectively. For universal quantification a tuple of e_1 belongs to the answer set if the value for A_1 does not overlap with the range of values that do not satisfy the predicate p.

Similar to Eqv. 4.4, we have to be careful when handling the special case $e_2 = \epsilon$: for universal quantification it is automatically evaluated to true. E.g. the aggregated value can be initialized to ∞ or $-\infty$ depending on $aggr$. In addition, we must be careful with the semantics of the aggregate function $aggr$ and the general comparison. The resulting unnested expression can be unnested further with rewrites of Sec. 4.5.

Equivalence 4.17

Preconditions Same preconditions as for Eqv. 4.15. But now predicate p can contain arbitrary boolean expressions.

Basic idea The θ-join is delegated to an ordinary join operator, which does not even have to be order-preserving. The outer antijoin preserves the order of the tuples in expression e_1.

4.4.4. Support Rewrites

Usually, we will have the same problems applying unnesting equivalences to universally quantified expressions as to existentially quantified ones: they may not be immediately applicable. Therefore, we need rules to rewrite universally quantified expressions, bringing them into the right shape. In general, we follow the same two strategies as in Section 4.3.4, reducing the number of free variables in a subexpression that is to be unnested and minimizing the distance between correlated query blocks.

$$\forall x \in e_1 : \forall y \in e_2 : p = \forall y \in e_2 : \forall x \in e_1 : p \quad (4.19)$$

$$\forall x \in e_1 : \neg p \vee q = \forall x \in (\sigma_p(e_1)) : q \quad (4.20)$$

$$p \wedge \forall x \in e_1 : q = \forall x \in e_1 : p \wedge q \quad (4.21)$$

$$\begin{aligned} \sigma_{\forall x \in e_2 : p \wedge \forall y \in e_3 : q}(e_1) &= \sigma_{\forall x \in e_2 : p}(\sigma_{\forall y \in e_3 : q}(e_1)) \\ &= \sigma_{\forall y \in e_3 : q}(\sigma_{\forall x \in e_2 : p}(e_1)) \end{aligned} \quad (4.22)$$

Figure 4.7.: Support rewrites for universally quantified queries

Let us now take a look at the rewrite rules (that are summarized in Figure 4.7). This is not a complete list, more rules can be found in the literature, e.g. [Bry89, JK84, Ste95].

Equivalence 4.19

Preconditions e_1 and e_2 can be evaluated independently of each other. Furthermore, Eqv. 4.19 was not applied to the same subexpression before.

Basic idea As with existential quantifiers, (independent) universal quantifiers can be exchanged (allowing the application of an unnesting rule that was not possible before).

Equivalence 4.20

Preconditions The rewrite has not been applied to the subexpression before.

Basic idea Depending on the direction in which we apply this equivalence in, we have the standard predicate push down or pull up (see also Eqv. 4.8 and [Bry89, JK84, Ste95]). Note that due to the universal quantifier, the predicate p has to be negated and is combined with the predicate q via a logical or-operator.

Equivalence 4.21

Preconditions The equivalence has not been applied to the same subexpression before. Also, the free variables in p are not bound by e_1 ($\mathcal{F}(p) \cap \mathcal{A}(e_1) = \emptyset$).

Basic idea We can freely move predicates that do not depend on attributes of an subexpression out of that expression.

Equivalence 4.22

Preconditions Eqv. 4.22 has not been applied to the same subexpression before. Also, all free variables in p and q are bound by the expressions e_1, e_2, or e_3 ($\mathcal{F}(p) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$ and $\mathcal{F}(q) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_3)$).

Basic idea Due to the order-preserving nature of the selection operator (and its commutativity), we are not bound to a specific evaluation order of selection operators.

4.4.5. Example Queries

As the overall strategies for unnesting universally quantified expressions are very similar to those for unnesting existential quantifiers, we restrict ourselves to three example queries.

Universal Quantification vs. Grouping

Similar to the second example query discussed in Section 4.3.5 universal quantification can be expressed in different ways. Besides the explicit quantified expression one can use function `fn::empty` or use counting. In the following example query we chose the quantified expression. It returns the authors whose books were all published after 1993.

```
for $a1 in distinct −values(doc("bib.xml")// author)
where every $b2 in doc("bib.xml")//book[author is $a1]
        satisfies $b2/@year gt 1993
return
  <new−author>
    { $a1 }
  <new−author>
```

During normalization we introduce a new **let** clause in the quantified expression to remove the path expression from the **satisfies** clause. We also move the path expression in the range expression into a new **let** clause. Finally, we move the comparison into a new **where** clause. These steps result in

```
for $a1 in distinct −values(doc("bib.xml")// author)
where every $b2 in doc("bib.xml")//book,
            $a2 in $b2/author
       let  $y2 := $b2/@year
      where $a1 is $a2
       satisfies $y2 gt 1993
return
  <new−author>
    { $a1 }
  <new−author>
```

The nested query plan is derived by application of the translation rules.

$$\Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,a1)}(\sigma_{\forall y \in \sigma_{a1=a2}(e2):y2>1993}(e_1)))$$

where

$$\begin{aligned} e_1 &= \Upsilon_{a1:\Pi^D(doc//author)}(\Box) \\ e_2 &= \chi_{y2:b2/@year}(\Upsilon_{a2:b2/author}(\Upsilon_{b2:doc//book}(\Box))) \end{aligned}$$

and

```
doc := doc("bib.xml")
s1 := "new-author"
```

Antijoin Eqv. 4.15 is applicable because the nested query contains a corelating predicate which performs an equality comparison and the range expression of the nested query can be evaluated independently. Then we can push the second part of the join predicate into its second operand.

$$\begin{aligned} & \Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,a1)}(\sigma_{\forall y \in \sigma_{a1=a2}(e2):y2>1993}(e_1))) \\ \overset{(4.15)}{=} \; & \Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,a1)}(e_1 \triangleright_{a1=a2 \wedge y2 \leq 1993} e_3)) \\ = \; & \Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,a1)}(e_1 \triangleright_{a1=a2} (\sigma_{y2 \leq 1993}(e_3)))) \end{aligned}$$

Grouping Since we know from the DTD that `author` elements occur only under `book` elements, $\Pi_{A_1}(e_1) = \Pi_{a1:a2}(\Pi_{a2}(e_2))$ holds and thus, we can apply Eqv. 4.18, which yields:

$$\begin{aligned} & \Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,a1)}(\sigma_{\forall y \in \sigma_{a1=a2}(e2):y2>1993}(e_1))) \\ \overset{(4.18)}{=} \quad & \Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,a1)}(\sigma_{c=0}(\Gamma_{c;=aa;count\circ\sigma_{y3\leq 1993}}(e_3)))) \end{aligned}$$

Notice that the steps taken in this example are similar to steps taken for the second example query in Section 4.3.5. It is quite easy to see that unnesting would work similar for a formulation of the current example query based on counting or the function `fn::empty` as discussed there.

Evaluation A comparison of the evaluation times of the discussed plans is given in the table below. The unnested query plans scale better than the nested plan because they need to scan the input document once or twice. In contrast to that the nested plan needs to execute the nested query as often as there are author elements in the input document.

Plan	100	1000	10000
Nested	0.12 s	4.86 s	507.85 s
Antijoin	0.07 s	0.08 s	0.24 s
Grouping	0.07 s	0.08 s	0.23 s

Non-Equality Correlating Predicates

Our second example query is an extension of the motivating query from the beginning of this section. In addition to checking the reserve price, we also make sure that a bid was placed in the specified period of time.

```
for $i in doc("items.xml")// itemtuple
where every $b in doc("bids.xml")// bidtuple
                                    [itemno eq $i/itemno]
      satisfies ($b/bid ge $i/ reserveprice
            and $b/ bid_date ge $i/ startdate
            and $b/ bid_date le $i/enddate)
return $i/itemno
```

The resulting normalized query looks as follows:

```
for $i in doc("items.xml")// itemtuple
let $ii := $i/itemno
let $ir := $i/ reserveprice
let $is := $i/ startdate
let $ie := $i/enddate
where every $b in doc("bids.xml")// bidtuple
      let $bi := $b/itemno
      let $bb := $b/bid
      let $bd := $b/biddate
      where $bi eq $ii
       satisfies ($bb ge $ir and
                  $bd ge $is and
                  $bd le $ie)
return $ii
```

After having normalized and translated this query, we arrive at the following algebraic expression. Again, we exploit the fact that child nodes occur exactly once.

$$\Pi_{ii}(\sigma_{\forall bt \in (\sigma_{bi=ii}(e_2)):bb \geq ir \wedge bd \geq is \wedge bd \leq ie}(e_1))$$

where

$$e_1 := \chi_{ie:i/enddate}(\chi_{is:i/startdate}(\chi_{ir:i/reserveprice}(\chi_{ii:i/itemno}(\Upsilon_{i:doc1//itemtuple}(\Box)))))$$

$$e_2 = \chi_{bd:b/biddate}(\chi_{bb:b/bid}(\chi_{bi:b/itemno}(\Upsilon_{b:doc2//bidtuple}(\Box))))$$

and

```
doc1 := doc("items.xml")
doc2 := doc("bids.xml")
```

Antijoin 1 Only one of the equivalences is immediately applicable: Eqv. 4.15. Applying this equivalence results in the following expression (note that the predicate $p = bb \geq ir \wedge bd \geq is \wedge bd \leq ie$ is negated for the antijoin):

$$\begin{aligned} & \Pi_{ii}(\sigma_{\forall bt \in (\sigma_{bi=ii}(e_2)):bb \geq ir \wedge bd \geq is \wedge bd \leq ie}(e_1)) \\ \overset{(4.15)}{=} \; & \Pi_{ii}(e_1 \triangleright_{bi=ii \wedge (bb<ir \vee bd<is \vee bd>ie)} e_2) \end{aligned}$$

Antijoin 2 Applying the support rewrite rule Eqv. 4.20 allows us to push down the predicate p. After that, we can merge it with the other selection and interpret the resulting predicate as a general θ-comparison, which matches the left hand side of Eqv. 4.17:

$$\begin{aligned} & \Pi_{ii}(\sigma_{\forall bt \in (\sigma_{bi=ii}(e_2)):(bb \geq ir \wedge bd \geq is \wedge bd \leq ie) \vee false}(e_1)) \\ \overset{(4.20)}{=} \; & \Pi_{ii}(\sigma_{\forall bt \in (\sigma_{bb<ir \vee bd<is \vee bd>ie}(\sigma_{bi=ii}(e_2))):false}(e_1)) \\ \overset{(4.17)}{=} \; & \Pi_{ii}(e_1 \triangleright_{ii=ii'} \Pi_{ii':ii}(e_1 \bowtie_{bi=ii \wedge (bb<ir \vee bd<is \vee bd>ie) \wedge true} e_2)) \end{aligned}$$

Evaluation The nested version of the query was implemented using a negated existential quantifier: $\Pi_{ii}(\sigma_{\nexists bt \in \sigma_{bi=ii}(e_2):bb<ir \vee bd<is \vee bd>ie}(e_1))$. This performs better because as soon as we find a tuple that satisfies the predicate, we can stop the evaluation of the nested query and return *false*.

In the table below, we present the execution times for the nested and the two unnested variants of the example query. As can be clearly seen, both unnested versions outperform the nested one:

Size	100	1000	10000
Nested	0.47s	11.39s	819.71s
Antijoin 1	0.21s	1.01s	8.54s
Antijoin 2	0.23s	1.68s	23.98s

Combining Existential and Universal Quantifiers

An interesting case that we have not looked at yet is mixing existentially and universally quantified expressions that are correlated with each other and the outer query block.[5] The following query returns the names of all users that bid on every item:

```
for $u in doc("users.xml")// usertuple
where every $i in doc("items.xml")// itemtuple
      satisfies some $b in doc("bids.xml")// bidtuple
                satisfies ($i/itemno eq $b/itemno and
                           $u/userid eq $b/userid)
return $u/name
```

During normalization we introduce variables and bind them to new **let** clauses.

[5] The result of the innermost existentially quantified expression depends on variable bindings passed by the two outer expressions. Notice that this query computes a relational division.

```
for  $u in doc("users.xml")// usertuple
let  $ui := $u/userid
let  $un := $u/name
where every $i in doc("items.xml")// itemtuple
        let    $in := $i/itemno
        satisfies some $b in doc("bids.xml")// bidtuple
                  let  $bn := $bb/itemno
                  let  $bi := $bb/userid
                  satisfies ($in eq $bn and
                             $ui eq $bi)
return $un
```

After having normalized and translated this query, we get the following algebraic expression:

$$\Pi_{un}(\sigma_{\forall it \in e_2 : \exists bt \in e_3 : in=bn \wedge ui=bi}(e_1))$$

where

$$\begin{aligned}
e_1 &:= \chi_{un:u/name}(\chi_{ui:u/userid}(\\
&\qquad \Upsilon_{u:doc1//usertuple}(\Box)))\\
e_2 &:= \chi_{in:i/itemno}(\Upsilon_{i:doc2//itemtuple}(\Box))\\
e_3 &:= \chi_{bi:b/userid}(\chi_{bn:b/itemno}(\Upsilon_{b:doc3//bidtuple}(\Box)))
\end{aligned}$$

and

```
doc1 := doc("users.xml")
doc2 := doc("items.xml")
doc3 := doc("bids.xml")
```

Antijoin When we try to unnest the translated query, we observe that we cannot use Eqv. 4.3 directly because the range predicate of the existential quantifier contains a quantified expression. We cannot apply Eqv. 4.15 either because the range predicate of the universal quantifier contains free variables that are not bound by the range expression of the universal quantifier.

We remedy this situation by pushing down the range predicates (once for the existential quantifier using Eqv. 4.8 and once for the universal quantifier using Eqv. 4.20). After that, we can unnest the inner query block by applying Eqv. 4.15 and then use Eqv. 4.13 for the final unnesting step (we use unnesting rules for universal quantifiers twice because by pushing down the existential quantifier we turn it into a universal quantifier):

$$\begin{aligned}
& \Pi_{un}(\sigma_{\forall it \in e_2 : \exists bt \in e_3 : in=bn \wedge ui=bi}(e_1))\\
\overset{(4.8)}{=}\ & \Pi_{un}(\sigma_{\forall it \in e_2 : (\exists bt \in \sigma_{in=bn}(e_3) : ui=bi) \vee false}(e_1))\\
\overset{(4.20)}{=}\ & \Pi_{un}(\sigma_{\forall it \in (\sigma_{\forall bt \in (\sigma_{in=bn}(e_3) : ui \neq bi)}(e_2)) : false}(e_1))\\
\overset{(4.15)}{=}\ & \Pi_{un}(\sigma_{\forall it \in (e_2 \triangleright_{in=bn \wedge ui=bi} e_3) : false}(e_1))\\
\overset{(4.13)}{=}\ & \Pi_{un}(e_1 \triangleright_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'} (\Pi_{\mathcal{A}(e_1)' : \mathcal{A}(e_1)}(\sigma_{true}(\Upsilon_{\mathcal{A}(e_2) : (e_2 \triangleright_{in=bn \wedge ui=bi} e_3)}(e_1)))))\\
=\ & \Pi_{un}(e_1 \triangleright_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'} (\Pi_{\mathcal{A}(e_1)' : \mathcal{A}(e_1)}(\Upsilon_{\mathcal{A}(e_2) : (e_2 \triangleright_{in=bn \wedge ui=bi} e_3)}(e_1))))
\end{aligned}$$

We compare two different unnested versions of the query. The first version is the expression above after applying Eqv. 4.15 (*Antijoin*). In the second version, we introduced an unnest map operator (*Antijoin + unnest map*). This version is more efficient, as we can stop evaluating the antijoin in the unnest map operator as soon as it produces a tuple (in that case, the current tuple of e_1 will be disqualified by the other antijoin operator).

We now discuss two alternative evaluation strategies also mentioned in [CKMP97]. The first evaluates the universal quantifier with counting and the second is based on relational division. We do not give algebraic equivalences because in an ordered context the unnesting approach presented above is most appropriate.

Grouping Another way to evaluate universal quantifiers works by counting matching tuples. When counting the number of non-matching tuples, we have to test for a count of zero. Alternatively the number of matching tuples must be equal to the number of tuples in the range expression.

$$\begin{aligned} & \Pi_{un}(\sigma_{g=count(e_2)}(\Gamma_{g;=p_1\cup\mathcal{A}(e_1);\Pi^{tid_{p_1}}_{t_1}\circ count}(\\ & \quad (tid_{p_1}(e_1) \times e_2) \ltimes_{in=bn\wedge ui=bi} e_3))) \\ = \quad & \Pi_{un}(\sigma_{g=count(e_2)}(\Gamma_{g;=p_1\cup\mathcal{A}(e_1);\Pi^{tid_{p_1}}_{t_1}\circ count}(\\ & \quad \Pi^{tid_{p_2}}(tid_{p_1}(e_1) \bowtie_{ui=bi} (tid_{p_2}(e_2) \bowtie_{in=bn} e_3))))) \end{aligned}$$

The idea of these plans is to detect, if any tuples of e_2 are discarted by the joins. Therefore, we compute the cardinality of e_2 using function `count`. This computation only incures a small overhead because we can count the number of tuples in e_2 while performing the join. We compare this value with the count computed for of each group in the grouping operator. Since the count value of each group is equal to the number of distinct matching tuples of e_2 with for each tuple in e_1, we can check that every tuple of e_2 actually found a join partner.

In the first plan (*Grouping 1*), we observe two things: (1) The argument of the grouping operator is a sequence of joins. Since order-preserving joins are associative we can explore an equivalent plan (*Grouping 2*). (2) We can avoid a costly cross product.

As we will see in the evaluation, the first plan is still the more efficient one. Nevertheless, only unnesting allows the cost-based decision between both plans.

Division Another alternative is based on the division operator. When we want to preserve the order, we either need to use nested-loop-based implementations or we need to sort after the division operator [GC95, RSMW02]. This decision should be made by the cost-based optimizer. Additionally, algebraic equivalences valid for division operators (as proposed for an algebra over sets [RM06]) become available after introducing the relational division operator. The resulting plans are as follows:

$$\begin{aligned} = \quad & \Pi_{un}(((e_1 \times e_2) \ltimes_{in=bn\wedge ui=bi} e_3) \div_{\mathcal{A}(e_2)} e_2) \\ = \quad & \Pi_{un}(\Pi^{tid_{p_1}}(tid_{p_1}(e_1) \bowtie_{ui=bi} (e_2 \bowtie_{in=bn} e_3)) \div_{\mathcal{A}(e_2)} e_2) \end{aligned}$$

We will refer to the first plan by *Division 1*. Again, we observe that we can exploit associativity of joins and replace the cross product by a join leading the the second plan (*Division 2*). Notice, that these plans introduce additional scans. Thus, we cannot expect more efficient execution plans. However, in an unordered context these plans might be efficient when operator implementations become available that can destroy order. We have looked at this possibility in plan *Division 3*:

$$\Pi_{un}(e_1 \ltimes ((e_1 \bowtie_{ui=bi} (e_2 \bowtie_{in=bn} e_3)) \div_{\mathcal{A}(e_2)} e_2)).$$

In this plan, the final semijoin filters all tuples in e_1 that do not qualify for the result and returns the result in document order.

Evaluation This is one of the rare cases where the unnested version of the query was not faster than the nested one. This underscores the importance of an algebraic approach in which different alternatives can be compared in a cost-based manner. The table below summarizes our experimental results for this example query:

Size	100	1000	10000
Nested	0.50s	11.12s	788.14s
Antijoin	0.31s	18.98s	2009.24s
Antijoin + unnest map	0.80s	15.18s	957.25s
Grouping 1	0.15s	8.38s	859.61
Grouping 2	9.74s	9730s	∞
Division 1	5.50s	5237s	∞
Division 2	6.78s	6434s	∞
Division 3	0.07s	0.12s	0.51

The fastest of the unnested plans based on order-presering operators are still in the same range of execution time. Obviously the alternative join order in the plans *Grouping 2* and *Division 2* is not better than the unnested plans with cross product and semijoin. The plan *Division 3*, which uses hash-based operator implementations for all operators, shows that improvements in orders of magnitudes become possible when order is discartend and reestablished later. Hence, in an unordered context unnested plans become much more efficient again. In [MHKM04] we have found similar results for join queries.

4.5. Implicit Grouping

Unlike SQL or OQL, which feature grouping clauses, XQuery does not have explicit grouping constructs yet. Grouping in XQuery is done via nested queries, hence we use the term *implicit grouping*. Although some researchers advocate introducing explicit grouping into XQuery [BC04, BCC+05], this does not mean that the option of implicit grouping will just vanish. Consequently, an optimization approach would have to be able to handle both cases. In the remainder of this section we present unnesting techniques for expressions containing implicit grouping.

4.5.1. Motivating Example

As a motivating example, we pick up the query from Section 3 again. In this query we rearrange all books such that they are grouped by their publishers:

```
for $p in distinct −values(doc("bib.xml")// publisher )
return
  <publisher>
        <name> { $p } </name>,
        { for $b in doc("bib.xml")//book[$p eq publisher ]
          return $b/ title
        }
  </publisher>
```

Recalling Section 3, we know that the normalization step for implicit grouping basically consisted of pulling up the **return** clause into a **let** clause and translating this **let** clause into a map operator. After having translated the normalized version of this query, we arrive at the basic pattern for implicit grouping (this time considering the **return** clause):

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(pn,t)}(\chi_{pn:\mathcal{C}(\text{elem},s2,p)}(\chi_{t:\Pi_{t2}(\sigma_{p=p2}(e_2))}(e_1)))))$$

where

$$e_1 := \Upsilon_{p:\Pi^D(doc//publisher)}(\Box)$$

$$e_2 := \chi_{t2:b/title}(\chi_{p2:b/publisher}(\Upsilon_{b:doc//book}(\Box)))$$

and

```
doc = doc("bib.xml")
s1  = "publisher"
s2  = "name"
```

We have now arrived at the standard pattern for implicit grouping. Strategies for unnesting this algebraic pattern can be found in the following section.

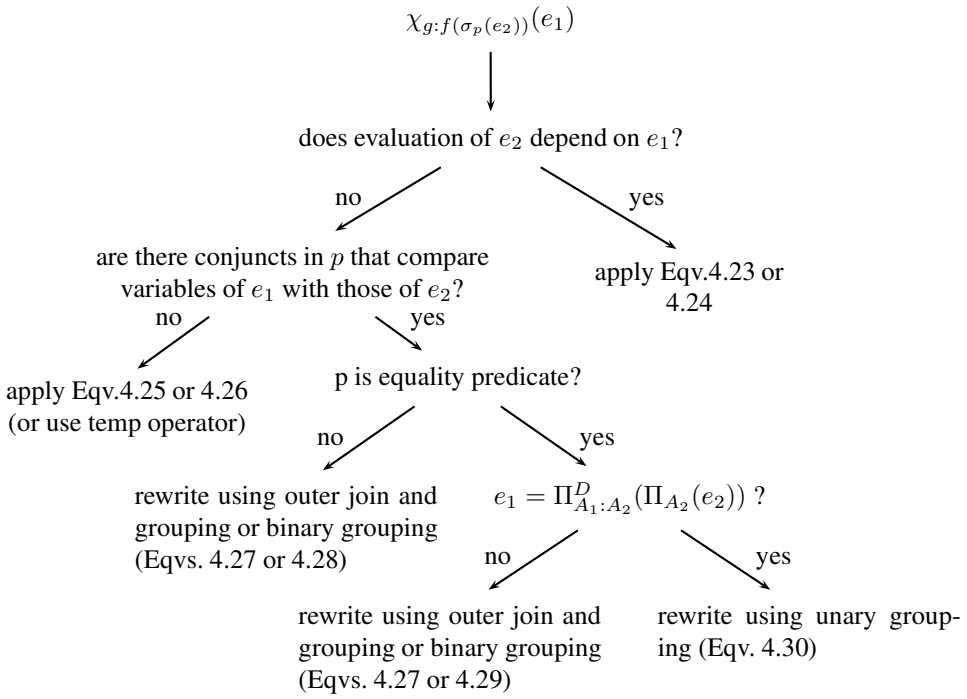


Figure 4.8.: Decision tree for implicit grouping, Equivalences and decisions refer to the case of value comparisons in predicates

4.5.2. Optimization Strategy

The strategy employed for unnesting expressions containing implicit grouping (see Figure 4.8 for an overview and Figure 4.9 for the equivalences) is similar to that for quantified expressions. First we check whether e_1 and e_2 can be evaluated independently of each other. If not, we have to rely on an unnest map operator. Otherwise, we take a look at the predicate p. Here, we distinguish the cases that e_1 and e_2

- are not correlated via p
- are correlated via a complex (non-equality) comparison operator
- are correlated via an equality predicate

For the equality predicate, there is room for further optimization if e_1 and e_2 produce identical sequences (save duplicates and additional attributes in e_2).

About our motivational example query we know the following: e_1 and e_2 can be evaluated independently, they are correlated via an equality predicate, and $e_1 = \Pi^D_{p:p2}(\Pi_{p2}(e_2))$. So we would apply Eqv. 4.30 in this case:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(pn,t)}(\chi_{pn:\mathcal{C}(\text{elem},s2,p)}(\\ \Pi_{p:p2}(\Gamma_{t;=p2;\Pi_{t2}}(\chi_{t2:b/title}(\chi_{p2:b/publisher}(\Upsilon_{b:doc//book}(\Box)))))))))$$

4.5.3. Equivalences for Unnesting

In Figure 4.9 the equivalences for unnesting implicit grouping can be found. As for unnesting quantified expressions before, we state the preconditions for applying an equivalence and give a brief description of the underlying idea. For most patterns we present two alternatives: one alternative that uses an outer join and unary grouping and another alternative

$$\chi_{g:f(\sigma_p(e_2))}(e_1) = e_1\Gamma_{g;\mathcal{A}(e_1)=A_1';f}(\Pi_{A_1':\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1))))) \quad (4.23)$$

$$\chi_{g:f(\sigma_p(e_2))}(e_1) = \Pi_{\overline{A_3}}(e_1 \bowtie^{g:f(\epsilon)}_{\mathcal{A}(e_1)=A_3} (\Pi_{A_3:\mathcal{A}(e_1)}(\Gamma_{g;=\mathcal{A}(e_1);f}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))))))) \quad (4.24)$$

$$\chi_{g:f(\sigma_p(e_2))}(e_1) = e_1\Gamma_{g;\mathcal{A}(e_1)=A_1';f}(\Pi_{A_1':\mathcal{A}(e_1)}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1) \times e_2))) \quad (4.25)$$

$$\chi_{g:f(\sigma_p(e_2))}(e_1) = \Pi_{\overline{A_3}}(e_1 \bowtie^{g:f(\epsilon)}_{\mathcal{A}(e_1)=A_3} (\Pi_{A_3:\mathcal{A}(e_1)}(\Gamma_{g;=\mathcal{A}(e_1);f}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1) \times e_2))))) \quad (4.26)$$

$$\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) = e_1\Gamma_{g;A_1\theta A_2;f}e_2 \quad (4.27)$$

$$\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) = \Pi_{\overline{A_3}}(e_1 \bowtie^{g:f(\epsilon)}_{A_1=A_3} (\Pi_{A_3:A_1}(\Gamma_{g;=A_1;f}(\Pi^D_{A_1}(e_1) \bowtie_{A_1\theta A_2} e_2)))) \quad (4.28)$$

$$\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) = \Pi_{\overline{A_2}}(e_1 \bowtie^{g:f(\epsilon)}_{A_1=A_2} (\Gamma_{g;=A_2;f}(e_2))) \quad (4.29)$$

$$\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) = \Pi_{A_1:A_2}(\Gamma_{g;=A_2;f}(e_2)) \quad (4.30)$$

Figure 4.9.: Unnesting equivalences for implicit grouping

that uses binary grouping. The first alternative uses operators that are more generally available in database systems, while the second alternative often results in more efficient plans. We will come back to this in our example queries.

Equivalence 4.23

Preconditions e_1 and e_2 cannot be evaluated independently (formally speaking, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$).

Basic idea For each tuple in e_1, we collect the corresponding tuples in e_2 via a binary grouping operator and apply the function f to all tuples in the corresponding group. We generate the tuples of the expression e_2 by combining all tuples t_1 in e_1 with all tuples in $e_2(t_1)$ via an unnest map operator and then apply p.

Equivalence 4.24

Preconditions e_1 and e_2 cannot be evaluated independently (formally speaking, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$).

Basic idea This is a variant of Eqv. 4.23. Instead of a binary grouping operator, we use a unary one. In order to avoid the "count bug" (i.e. losing a tuple due to an empty group) we use an outer join operator. The main motivation for this variant is the fact that not every DBMS supports a binary grouping operator.

Equivalence 4.25

Preconditions e_1 and e_2 can be evaluated independently ($\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$).

Basic idea This equivalence looks very similar to Eqv. 4.23 except that e_2 can be evaluated independently of e_1 and, therefore, is connected via a Cartesian product to each tuple in e_1. For each tuple in e_1, the tuples in e_2 are grouped via a binary grouping operator. If the predicate p does not refer to attributes in e_1, we could also compute $f(\sigma_p(e_2))$, store the result temporarily, and attach this result to each tuple in e_1 (as in this case, we have the same group for each tuple in e_1).

Equivalence 4.26

Preconditions e_1 and e_2 can be evaluated independently ($\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$).

Basic idea This is the outer join/unary grouping variant of Eqv. 4.25.

Equivalence 4.27

Preconditions e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with a predicate containing a θ-comparison.

Basic idea As we know more about the attributes involved in the predicate, we can group the tuples in e_2 directly without connecting them to tuples in e_1 first. The predicate correlating e_1 and e_2 is now an element of the binary grouping operator.

Equivalence 4.28

Preconditions e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with a predicate containing a θ-comparison.

Basic idea This is the outer join/unary grouping variant of Eqv. 4.27. The elegant integration of the correlating predicate into the grouping operator is not possible here, as we use a unary grouping operator. So this looks more like Eqv. 4.26, replacing the cross product with a θ-join. This technique is also known as *magic set decorrelation* [SPL96]. (The θ-join between e_1 and e_2 needs only be order-preserving if the correct computation of f relies on ordered tuples.)

Equivalence 4.29

Preconditions e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with an equality predicate.

Basic idea In the special case of an equality predicate, the function f is computed for each possible group identified in e_2. The main advantage is that the result of the grouping needs only be evaluated once and can be materialized. The variant using a binary grouping operator is already covered by Eqv. 4.27.

Equivalence 4.30

Preconditions e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with an equality predicate. Also, $e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$, assuming that $A_i = \mathcal{A}(e_i)$.

Basic idea If we know that there are no empty groups (because e_1 and e_2 contain the same attribute values, save attribute names and duplicates), we do not need to evaluate e_1, but can do a unary grouping on e_2.

4.5.4. Support Rewrites

$$\Pi_{\overline{g_1}}(\chi_{g_2:f(g_1)}(\chi_{g_1:e_2}(e_1))) = \chi_{g2:f(e_2)}(e_1) \quad (4.31)$$

$$\Upsilon_{A:\Upsilon_{B:e_2}(\Box)}(e_1) = \Upsilon_{A:e_2}(e_1) \quad (4.32)$$

$$\Pi^{tid_B}_{A_i}(tid_B(e_1 \times e_2)) = \Pi^{tid_{B_1},tid_{B_2}}_{A_i}(tid_{B_1}(e_1) \times tid_{B_2}(e_2)) \quad (4.33)$$

$$\Pi^{tid_B}_{A_1}(\Pi^{tid_C}_{A_1}(e_1)) = \Pi^{tid_B}_{A_1}(e_1) \quad (4.34)$$

Figure 4.10.: Support Rewrites

As the unnesting equivalences from Section 4.5.3 expect certain patterns, we may have to rewrite nested algebraic expressions to match these patterns. Figure 4.10 gives a quick

overview of the support rewrite rules for unnesting implicit grouping. The underlying ideas are explained in the following:

Equivalence 4.31

Preconditions None

Basic idea This rewrite merges two map operators into one and can be used when the result of one map operator is just consumed by another map operator and does not appear anywhere else afterwards. This is useful, as it saves us from constructing the (possibly sequence-valued) attribute g_1.

Equivalence 4.32

Preconditions None

Basic idea This rewrite merges two unnest map operators into one. We eliminate an unnecessary step of nesting and then unnesting again.

Equivalence 4.33

Preconditions None

Basic idea We break up a tid operator that assigns a unique id to each tuple of a Cartesian product into two tid operators operating on the subexpressions of the product. We can do this because each tuple of the cross product is still identifiable as before. When discarding duplicates, we have to look at both tids. This rewrite allows us to push down operators into the cross product (e.g. selections turning the product into a join).

Equivalence 4.34

Preconditions The tids are assigned in such a way in e_1 that the attribute B is functionally dependent on C ($C \rightarrow B$).

Basic idea In this case, we can get rid of the inner duplicate elimination, as each tuple that is filtered out by $\Pi^{tid_C}_{A_1}$ will also be filtered out by $\Pi^{tid_B}_{A_1}$.

Queries with implicit grouping involving general comparison operators are handled by transforming them into existentially quantified expressions during normalization. It follows that all equivalences (unnesting and support rewrite) found in Section 4.3 can also be used as support rewrite rules when unnesting implicit grouping expressions containing general comparisons.

4.5.5. Example Queries

Let us now show how to apply the unnesting equivalences to concrete example queries. First, we present two simple example queries for detecting grouping with aggregation in the **return** clause or in the **where** clause. After that we discuss several examples that are more involved. In the third example query we investigate how we detect a unary grouping operator. Then, we discuss combining the unnesting rules for grouping with those for quantified expressions to demonstrate the full power of our framework. Depending on whether the variables used in our queries are atomic or sequence-valued, we have to employ a value-based or a general comparison operator. We distinguish between the variables in the outer query block and those in the inner (implicit grouping) query block. As both sets of variables can be atomic or sequence-valued, we have four different cases. For each of these cases we will present an example query and discuss its optimization.

Aggregation

Aggregation is often used in conjunction with grouping. In this query we want to find the minimal price for each book which is identified by its title.

```
for $t1 in distinct −values(doc("prices.xml")//book/title)
let $p1 := for $p2 in doc("prices.xml")//book
                                    [ title eq $t1 ]/price
           return decimal($p2)
return
  <minprice title ="{ $t1 }">
    <price> { min($p1) } </price>
  </minprice>
```

We first normalize the query. In general, we have to be very careful when rewriting a path expression. Breaking up the XPath expression in the query is only possible because we know from the DTD that every `book` element has exactly one `price` child element and exactly on `title` child element. We also move the element construction into new **let** clauses.

```
for $t1 in distinct −values(doc("prices.xml")//book/title)
let $p1 := (for $b2 in doc("prices.xml")//book
            let $t2 := $b2/title,
                $p2 := $b2/price,
                $c2 := decimal($p2)
            where $t1 eq $t2
            return $c2),
    $m1 := min($p1),
    $pt := <price> { $m1 } </price>,
    $res := <minprice title ="{ $t1 }"> { $pt } </minprice>
return $res
```

After these rather complex normalization steps translation is straight forward

$$\Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,ra,pt)}(\chi_{ra:\mathcal{C}(attr,s2,t1)}(\chi_{pt:\mathcal{C}(elem,s3,m1)}(\chi_{m1:min(p1)}(\chi_{p1:\Pi_{c2}(\sigma_{t1=t2}(e_2))}(e_1))))))$$

where

$$e_1 = \Upsilon_{t1:\Pi^D(doc//book/title)}(\Box)$$

$$e_2 = \chi_{c2:decimal(p2)}(\chi_{p2:b2/price}(\chi_{t2:b2/title}(\Upsilon_{b2:doc//book}(\Box))))$$

and

```
doc := doc("prices.xml")
s1  := "minprice"
s2  := "title"
s3  := "price"
```

The translated query contains a rather complex sequence of node constructors. Since we focus on query unnesting here, we define

$$\Xi_{res}(\dots) := \Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,ra,pt)}(\chi_{ra:\mathcal{C}(attr,s2,t1)}(\chi_{pt:\mathcal{C}(elem,s3,m1)}(\dots))))$$

as an abreviation.

Unnesting We start with merging the map operator containing the nested query with the computation of the minimum using Eqv. 4.31. Thereby, we avoid materializing a sequence-valued result and at the same time remove a variable binding that is subsequently not used any more. Since only `title` elements under `book` elements are considered, not only are Eqvs. 4.27 and 4.29 applicable but the restriction $e_1 = \Pi_{t1:t2}(\Pi^D_{t2}(e_2))$ holds and Eqv. 4.30 can be used. As we will see in our third example query, the latter results in the most efficient

plan. Hence, we neglect the other possibilities. Applying Eqv. 4.30 leaves us with

$$\begin{aligned} & \Xi_{res}(\chi_{m1:min(p1)}(\chi_{p1:\Pi_{c2}(\sigma_{t1=t2}(e_2))}(e_1))) \\ \overset{(4.31)}{=} \quad & \Xi_{res}(\chi_{m1:min(\Pi_{c2}(\sigma_{t1=t2}(e_2)))}(e_1)) \\ \overset{(4.30)}{=} \quad & \Xi_{res}(\Pi_{t1:t2}(\Gamma_{m1;=t2;min\circ\Pi_{c2}}(e_2))). \end{aligned}$$

Evaluation Below, we compare the evaluation times for the two plans. While the nested plan needs to scan the document $|book| + 1$ times, the unnested plan unsing grouping needs to scan the document just one time. Here $|book|$ is the number of book elements in the input document, i.e. 100, 1000, or 10000 books. The measurements demonstrate the massive performance improvements as an immediate consequence.

Size	100	1000	10000
Nested	0.09 s	1.81 s	173.51 s
Grouping	0.07 s	0.08 s	0.19 s

Aggregation in the Where Clause

Let us consider a query where nesting occurs in a predicate in the **where** clause that depends on an aggregate function, `count` in this case. This is similar to a having-clause in SQL: after grouping bids by `itemno`, they are selected by the result of the aggregation. The query returns all popular items offered, i.e. all items with at least three bids.

```
let $d1 := doc("bids.xml")
for $i1 in distinct-values($d1//itemno)
where count($d1// bidtuple [itemno eq $i1 ]) ge 3
return $i1
```

During normalization we extract the left argument of the value comparison, turn it into a **let** clause, and move the XPath predicate into a **where** clause.

```
let $d1 := doc("bids.xml")
for $i1 in distinct-values($d1//itemno)
let $i3 := (for $i2 in $d1// bidtuple /itemno
            where $i1 = $i2
            return $i2)
let $c1 := count($i3)
where $c1 ge 3
return $i1
```

Now the translation into our algebra is easy.

$$\Pi_{i1}(\sigma_{c1\geq 3}(\chi_{c1:count(i3)}(\chi_{i3:\Pi_{i2}(\sigma_{i1=i2}(e_2))}(e_1))))$$

where

$$\begin{aligned} e_1 &:= \Upsilon_{i1:\Pi^D(d1//itemno)}(\chi_{d1:doc}(\Box)) \\ e_2 &:= \Upsilon_{i2:d2//bidtuple/itemno}(\Box) \end{aligned}$$

and

```
doc := doc("bids.xml")
```

Unnesting We would like to apply Eqv. 4.30 for unnesting the above expression. In order to do that, we have to check that the prerequisites hold. Looking at the DTD of bids.xml, we see that `itemno` elements appear only directly beneath `bidtuple` elements. Thus, the condition $e_1 = \Pi_{i1:i2}(\Pi^D_{i2}(e_2))$ holds, and we can apply Eqv. 4.30. Again we merge two adjacent map operators before we apply the unnesting equivalence.

$$\begin{aligned} & \Pi_{i1}(\sigma_{c1\geq 3}(\chi_{c1:count(i3)}(\chi_{i3:\Pi_{i2}(\sigma_{i1=i2}(e_2))}(e_1)))) \\ \overset{(4.31)}{=} \; & \Pi_{i1}(\sigma_{c1\geq 3}(\chi_{c1:count(\Pi_{i2}(\sigma_{i1=i2}(e_2)))}(e_1))) \\ \overset{(4.30)}{=} \; & \Pi_{i1}(\sigma_{c1\geq 3}(\Pi_{i1:i2}(\Gamma_{c1;=i2;count}(e2)))) \end{aligned}$$

Evaluation The evaluation times for each plan are given in the table below. The number of bids — shown as column heading in the table below — and items is varied. The number of items equals 1/5 times the number of bids. Again, the measurements verify the effectiveness of the unnesting techniques.

Size	100	1000	10000
Nested	0.06 s	0.53 s	48.1 s
Grouping	0.06 s	0.07 s	0.10 s

Unary Grouping

We have looked at two rather simple queries. Now we discuss more involved queries. The query below restructures the input document by grouping books by authors. Its result contains for each author a sequence of book title. In contrast to the previous examples, these book titles are not summarized into an aggregated value.

```
let $d1 := doc("bib.xml")
for $a1 in distinct −values($d1//author)
return
  <author>
        <name> { $a1 } </name>
        {
          for $b2 in $d1/book[$a1 = author]
          return $b2/ title
        }
  </author>
```

Normalization of the query first moves the nested FLWR expression outside the **return** clause into a new **let** clause. We prepare the moved **for** clause for the translation into an algebraic expression by introducing new variables. We further move the predicate at the end of the path expression into the **where** clause. Since the predicate performs a general comparison we turn this comparison into a quantified query. Thereby the existential semantics of this predicate are made explicit.

```
let $d1 := doc("bib.xml")
for $a1 in distinct −values($d1//author)
let $t1 := (for $b2 in $d1/book
              let $t2 := $b2/ title
              where some $a2 in $b2/author
                    satisfies $a1 eq $a2
              return $t2)
let $an := <name> { $a1 } </name>
let $res := <author> { $an, $t1 } </author>
return $res
```

From the DTD we know that every book contains only a single `title` element. Hence, the projection on $t2$ returns a sequence of those elements. We do not have to take care of implicit flattening of nested sequences in the `return` clause of the inner query block. The translation then results in

$$\Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,an,t1)}(\chi_{an:\mathcal{C}(elem,s2,a1)}(\chi_{t1:\Pi_{t2}(\sigma_{\exists a3\in\Upsilon_{a2:b2/author}(\square):a1=a2}(e_2))}(e_1))))$$

where

$$e_1 := \Upsilon_{a1:\Pi^D(d1//author)}(\chi_{d1:doc}(\Box))$$
$$e_2 := \chi_{t2:b2/title}(\Upsilon_{b2:d1/book}(\Box))$$

and

```
doc := doc("bib.xml")
s1  := "author"
s2  := "name"
```

To avoid clutter and since we focus on query unnesting here, we define

$$\Xi_{res}(\dots) := \Pi_{res}(\chi_{res:\mathcal{C}(elem,s1,an,t1)}(\chi_{an:\mathcal{C}(elem,s2,a1)}(\dots)))$$

as an abreviation.

Unnest Existential Quantifier Our goal is to apply any of our unnesting equivalences and explicitly compute the groups using a grouping operator. Before we can do that we have to remove the existential quantifier because our equivalences test for value comparisons as correlating predicates.

Clearly, the evaluation of the range expression of the quantifier depends on its enclosing block. Hence, we have to apply Eqv. 4.1 to unnest the nested query. After that we can simplify the resulting expression by merging projections and unnest map operators (Eqv. 4.32). This step establishes the basic pattern for implicit grouping.

$$\begin{aligned}
& \Xi_{res}(\chi_{t1:\Pi_{t2}(\sigma_{\exists a3\in\Upsilon_{a2:b2/author}(\Box):a1=a2}(e_2))}(e_1)) \\
\overset{(4.1)}{=}\ & \Xi_{res}(\chi_{t1:\Pi_{t2}(\Pi^{tid_{t3}}_{\mathcal{A}(e_2)}(\sigma_{a1=a2}(\Upsilon_{\Upsilon_{a2:b2/author}(\Box)}(tid_{t3}(e_2)))))}(e_1)) \\
\overset{(4.32)}{=}\ & \Xi_{res}(\chi_{t1:\Pi^{tid_{t3}}_{t2}(\sigma_{a1=a2}(\Upsilon_{a2:b2/author}(tid_{t3}(e_2))))}(e_1))
\end{aligned}$$

Binary Grouping Looking at result of the previous steps, Eqv. 4.27 is an obvious candidate to unnest this algebraic expression resulting in:

$$\overset{(4.27)}{=}\ \Xi_{res}(e_1\Gamma_{t1;a1=a2;\Pi^{tid_{t3}}_{t2}}(\Upsilon_{a2:b2/author}(tid_{t3}(e_2))))$$

Outer Join According to the decision tree presented in Figure 4.8 Eqv. 4.29 is annother candidate resulting in:

$$\overset{(4.29)}{=}\ \Xi_{res}(\Pi_{\mathcal{A}(e_1)\cup t1}(e_1 \ \mathbb{M}^{t1:\epsilon}_{a1=a2}\ (\Gamma_{t1;=a2;\Pi^{tid_{t3}}_{t2}}(\Upsilon_{a2:b2/author}(tid_{t3}(e_2))))))$$

For these two unnested plans we can expect drastic improvements. Despite the fact that we can unnest this query to an even more efficent plan, we include them here for two reasons. First, the condition for applying Eqv. 4.30 can be hard to verify. Hence, we expect that unnesting often results in plans using one of the two alternative plans above. Second, we want to investigate the performance differences at query execution time of the three alternatives we will discuss here. Particularly, we can decide how much performance loss we suffer when we cannot detect that Eqv. 4.30 is applicable and wether we should prefer binary grouping to unary grouping and outer join.

Unary Grouping Looking more closely at the nested algebraic expression after simplification, we realize that Eqv. 4.30 is also applicable. In order to meet the conditions of Eqv. 4.30, we have to verify that $e_1 = \Pi_{a_1:a_2}(\Pi^D_{a2}(\Upsilon_{a2:b2/author}(tid_{t3}(e_2))))$ holds. This is indeed the case if there are no `author` elements other than those directly under `book` elements. This is the case for the DTD given for document bib.xml, and we can apply this equivalence:

$$\overset{(4.30)}{=} \quad \Xi_{res}(\Pi_{a1:a2}(\Gamma_{t1;=a2;\Pi^{tid_{t3}}_{t2}}(\Upsilon_{a2:b2/author}(tid_{t3}(e_2)))))$$

Note that although the order is destroyed on authors, all unnested expressions produce the titles of each author in document order, as is required by the XQuery semantics for this query.

Evaluation In the table below, we summarize the evaluation times for the first query. The document `bib.xml` contained either 100, 1000, or 10000 books with ten authors per book.

Size	100	1000	10000
Nested	0.40 s	31.65 s	3195 s
Binary Grouping	0.12 s	0.32 s	2.45 s
Outer Join	0.13 s	0.33 s	3.31 s
Unary Grouping	0.12 s	0.32 s	1.85 s

The nested plan needs to scan the document $|author| + 1$ times where $|author|$ is the number of author elements in the input document. The query plans using either binary grouping or the outer join need to scan the input document twice. For this query, binary grouping performs faster then unary grouping and outerjoin. But since the last plan performes just one scan it is always the fastest. Nevertheless, the improvement is rather small compared to the effect of unnesting into either unnested plan.

Non-Equality Correlating Predicates

The following example query counts the number of bids for each item where the `reserveprice` for the item is less than the price of the bid. In this case, all variables are atomic.

```
for $i in doc("items.xml")// itemtuple
return
  <item>
    { $i/itemno },
    <count> { count(for $b in doc("bids.xml")// bidtuple
                    where $i/ reserveprice lt $b/bid
                      and $i/itemno eq $b/itemno
                    return $b) }
    </count>
  </item>
```

The normalization step introduces several new **let** clauses, pulling up the nested **return** clause and moving path expressions:

```
for $i in doc("items.xml")// itemtuple
let $in := $i/itemno
let $ir := $i/ reserveprice
let $bt := (for $b in doc("bids.xml")// bidtuple
            let $bn := $b/itemno
            let $bb := $b/bid
            where $ir lt $bb and $in eq $bn
            return $b)
let $ct := count($bt)
let $ce := <count> { $ct } </count>
let $sq := ($in, $ct)
let $res := <item> { $sq } </item>
return $res
```

Translating this into our algebra is now straightforward:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(in,ce)}(\chi_{ce:\mathcal{C}(\text{elem},s2,ct)}(\chi_{ct:count(bt)}(\chi_{bt:(\Pi_b(\sigma_{ir<bb\wedge in=bn}(e_2)))}(e_1))))))$$

where

$$\begin{aligned} e_1 &:= \chi_{ir:i/reserveprice}(\chi_{in:i/itemno}(\Upsilon_{i:doc1//itemtuple}(\Box))) \\ e_2 &:= \chi_{bb:b/bid}(\chi_{bn:b/itemno}(\Upsilon_{b:doc2//bidtuple}(\Box))) \end{aligned}$$

and

```
doc1 := doc("items.xml")
doc2 := doc("bids.xml")
s1   := "item"
s2   := "count"
```

Binary Grouping The first step in unnesting this query is to combine map operators via rewrite rule 4.31 (we can do this because, e.g. the attribute bt created by the inner map operator is not needed in the remainder of the algebraic expression). After that, we have reached our standard pattern for implicit grouping and can apply Eqv. 4.27, as e_1 and e_2 can be evaluated independently and are correlated via a non-equality predicate:

$$\begin{aligned} &\overset{(4.31)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(in,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_b(\sigma_{ir<bb\wedge in=bn}(e_2)))}(e_1))) \\ &\overset{(4.27)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(in,\mathcal{C}(\text{elem},s2,ct)))}(e_1 \Gamma_{ct;ir<bb\wedge in=bn;count\circ\Pi_b} e_2)) \end{aligned}$$

Outer Join After having merged the two map operators we can also apply the alternative equivalence 4.28 using an outer join and a unary grouping operator:

$$\begin{aligned} \overset{(4.28)}{=}\ & \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(in,\mathcal{C}(\text{elem},s2,ct)))}(e_1 \ \mathbb{M}^{ct:0}_{in=in'\wedge ir=ir'} \\ & (\Pi_{in':in,ir':ir}(\Gamma_{ct;=in,ir;count}(\Pi^D_{in,ir}(e_1) \bowtie_{in=bn\wedge bb>ir} e_2))))) \end{aligned}$$

Evaluation As can be clearly seen in following table, both unnested versions of the query outperform the nested version by orders of magnitude. For the expression involving the binary grouping operator, we used our implementation as presented in [MM05a], which in this case is more efficient than the outer join expression.

Size	100	1000	10000
Nested	0.12s	10.22s	1008.17s
Binary Grouping	0.10s	0.16s	0.72s
Outer Join	0.11s	0.22s	3.08s

Sequence-Valued Attribute in Nested Expression

The following query counts for each author the number of times he or she has been an editor. This query features implicit grouping with a sequence-valued comparison in the nested subexpression. We transform this into an existentially quantified subquery during normalization.

```
for $a in distinct −values(doc("bib.xml")//book/author)
return
  <author−editor>
    { $a },
    <count> { count(for $c in doc("bib.xml")//book
                    where $a = $c/editor
                    return $c)
    } </count>
  </author−editor>
```

Normalization introduces the usual **let** clauses for the implicit grouping and path expressions. The general comparison is turned into a nested quantified query containing a FLWR expression:

```
for $a in distinct −values(doc("bib.xml")//book/author)
let $ae := (for $c in doc("bib.xml")//book
            let $ce := $c/editor
            where some $ce1 in $ce
                  satisfies $a eq $ce1
            return $c)
let $ct := count($ae)
let $ci := <count> { $ct } </count>
let $sq := ($a, $ci)
let $res := <author−editor> { $sq } </author−editor>
return $res
```

The translation of the normalized query into our algebra yields:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(a,ci)}(\chi_{ci:\mathcal{C}(\text{elem},s2,ct)}(\chi_{ct:count(ae)}(\chi_{ae:\Pi_c(\sigma_{\exists cet\in\Upsilon_{ce1:ce(\square):a=ce1}(e_2)})}(e_1))))))$$

where

$$e_1 := \Pi^D(\Upsilon_{a:doc//book/author}(\square))$$
$$e_2 := \chi_{ce:c/editor}(\Upsilon_{c:doc//book}(\square))$$

and

```
doc := doc("bib.xml")
s1  := "author-editor"
s2  := "count"
```

Binary Grouping In a first step, we unnest the nested expression introduced by the existential quantifier. As the range expression of the selection depends on e_2, we have to apply Eqv. 4.1. After that, we merge two map and two unnest map operators (Eqv. 4.31 and Eqv. 4.32, respectively). Finally, we apply an unnesting rule for implicit grouping (Eqv. 4.27).

$$\overset{(4.1)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(a,ci)}(\chi_{ci:\mathcal{C}(\text{elem},s2,ct)}(\chi_{ct:count(ae)}(\chi_{ae:\Pi_c(\Pi^{tid_B}_{\mathcal{A}(e_2)\cup ce1}(\sigma_{a=ce1}(\Upsilon_{ce1:\Upsilon_{ce1:ce(\square)}(tid_B(e_2))}))))}(e_1))))))$$

$$\overset{(4.31)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_c(\Pi^{tid_B}_{\mathcal{A}(e_2)\cup ce1}(\sigma_{a=ce1}(\Upsilon_{ce1:\Upsilon_{ce1:ce(\square)}(tid_B(e_2))}))))}(e_1)))$$

$$\overset{(4.32)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_c(\Pi^{tid_B}_{\mathcal{A}(e_2)\cup ce1}(\sigma_{a=ce1}(\Upsilon_{ce1:ce}(tid_B(e_2))))))}(e_1)))$$

$$\overset{(4.27)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;a=ce1;count\circ\Pi_c\circ\Pi^{tid_B}_{\mathcal{A}(e_2)\cup ce1}}\Upsilon_{ce1:ce}(tid_B(e_2))))$$

Outer Join In the last unnesting step, we can also use the alternative equivalence based on outer join and unary grouping. Since we have a correlating predicate based on equality, we can apply Eqv. 4.29:

$$\overset{(4.29)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,a,\mathcal{C}(\text{elem},s2,ct))}(e_1 \bowtie^{ct:0}_{a=ce1} (\Gamma_{ct;=ce1;count\circ\Pi_c\circ\Pi^{tid_B}_{\mathcal{A}(e_2)\cup ce1}}\Upsilon_{ce1:ce}(tid_B(e_2)))))$$

Evaluation The result for this query looks very similar to the results for the query from the previous section. Overall, the running times are larger due to the general comparison in the nested expression. Again, the implementation based on the binary grouping is faster than the one based on the outer join operator (however, this time the difference is significant).

Size	100	1000	10000
Nested	7.08s	655.66s	∞
Binary Grouping	0.16s	0.73s	6.63s
Outer Join	0.20s	2.34s	415.26s

Sequence-Valued Attribute in Outer Expression

In this section we present an example query in which the sequence-valued attribute is located in the outer query block. For each author we count the number of books that are cheaper than any book written by that particular author.

The main difficulties in evaluating this query efficiently are the following. The values that we group on (i.e. the authors) are not found in the correlating predicate (cf. [BCC+04, BCC+05] on the grouping problem). In addition to that, the groups are created based on a non-equality predicate.

```
for $a in distinct−values(doc("bib.xml")//book/author)
let $ap := doc("bib.xml")//book[$a = author ]/ price
return
  <cheaper−books>
    { $a },
    <count>
      { count(doc("bib.xml")//book[price < $ap]) }
    </count>
  </cheaper−books>
```

During normalization we turn the XPath predicates into **where** clauses in FLWR expressions. This is correct because the comparisons always return boolean values so that these predicates cannot result in positional predicates during evaluation. And books without price are handled properly during the construction of the sequence bound to variable ap. From the DTD, we know that each book always has exactly one price. Normalization introduces several new **let** expressions and shifts the implicit grouping out of the **return** block.

```
for $a in distinct−values(doc("bib.xml")//book/author)
let $ap := (for $ab in doc("bib.xml")//book
            let $aba := $ab/author
            let $abp := $ab/price
            where some $aa in $aba
                  satisfies $a eq $aa
            return $abp)
let $lp := (for $pb in doc("bib.xml")//book
            let $pp := $pb/price
            where some $pa in $ap
                  satisfies $pp lt $pa
            return $pb)
let $ct := count($lp)
let $ce := <count> { $ct } </count>
let $sq := ($a, $ce)
let $res := <cheaper−books> { $sq } </cheaper−books>
return $res
```

The translation step produces the following algebraic expression:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(a,ce)}(\chi_{ce:\mathcal{C}(\text{elem},s2,ct)}(\chi_{ct:count(lp)}(\chi_{lp:\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap(\square):pp<pa}(e_3)})}(\chi_{ap:\Pi_{abp}(\sigma_{\exists at\in\Upsilon_{aa:aba(\square):a=aa}(e_2)})}(e_1)))))))$$

where

$$\begin{aligned} e_1 &:= \Pi^D(\Upsilon_{a:doc//book/author}(\square)) \\ e_2 &:= \chi_{abp:ab/price}(\chi_{aba:ab/author}(\Upsilon_{ab:doc//book}(\square))) \\ e_3 &:= \chi_{pp:pb/price}(\Upsilon_{pb:doc//book}(\square)) \end{aligned}$$

and

```
doc := doc("bib.xml")
s1  := "cheaper-books"
s2  := "count"
```

Binary Grouping Unnesting this algebraic expression involves several steps. First, we unnest the inner existentially quantified expression (applying Eqv. 4.1 as the range predicate depends on e_2). After that, we eliminate a redundant unnest map operator using the support rewrite rule 4.32. Then we are ready to apply an equivalence for unnesting grouping on the inner map operator. Eqv. 4.30 is the most efficient variant in this case, resulting in a unary grouping operator on e_2:

$$\begin{aligned}
&\overset{(4.1)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\\
&\quad \chi_{ct:count(lp)}(\chi_{lp:\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap(\square):pp<pa}(e_3)})}(\\
&\quad \chi_{ap:\Pi_{abp}(\Pi^{tid_B}_{\mathcal{A}(e_2)\cup aa}(\sigma_{aa=a}(\Upsilon_{aa:\Upsilon_{aa:aba(\square)}(tid_B(e_2))}(e_1))))))) \\
&\overset{(4.32)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(lp)}(\chi_{lp:\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap(\square):pp<pa}(e_3)})}(\\
&\quad \chi_{ap:\Pi_{abp}(\Pi^{tid_B}_{\mathcal{A}(e_2)\cup aa}(\sigma_{aa=a}(\Upsilon_{aa:aba}(tid_B(e_2))))}(e_1))))) \\
&\overset{(4.30)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(lp)}(\chi_{lp:\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap(\square):pp<pa}(e_3)})}(\\
&\quad \underbrace{\Pi_{a:aa}(\Gamma_{ap;=aa;\Pi_{abp}\circ\Pi^{tid_B}_{\mathcal{A}(e_2)\cup aa}}\Upsilon_{aa:aba}(tid_B(e_2))))))}_{e_4})
\end{aligned}$$

In order to keep things readable, we call the inner, unnested expression e_4 in the following. We continue by merging the two remaining map operators via Eqv. 4.31, prepare the existentially quantified subexpression for unnesting using Eqv. 4.8, and then unnest it by applying Eqv. 4.4. As mentioned earlier, we have to be careful with the semantics of function max. In our query it has to use string comparison to compute the maximum:

$$\begin{aligned}
&\overset{(4.31)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap(\square):pp<pa}(e_3)}))}(e_4))) \\
&\overset{(4.8)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_{pb}(\sigma_{\exists pt\in\sigma_{pp<pa}(\Upsilon_{pa:ap(\square)})(e_3)}))}(e_4))) \\
&\overset{(4.4)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_{pb}(\sigma_{pp<\max_{pa}(\Upsilon_{pa:ap(\square)})(e_3)}))}(e_4)))
\end{aligned}$$

Finally, we are now ready to unnest the grouping expression containing the count-function. Here we use the variant based on binary grouping (the outer join/unary grouping variant will be presented in just a moment). After having unnested the expression, we can transform the unnest map operator into a Cartesian product, as the two involved expressions can be evaluated independently of each other. In a last step, we change the selection and cross

product into a join operator:

$$
\begin{aligned}
&\overset{(4.23)}{=} \quad \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\\
&\qquad e_4\Gamma_{ct;\mathcal{A}(e_4)=A'_4;count\circ\Pi_{pb}}\Pi_{A'_4:\mathcal{A}(e_4)}(\sigma_{pp<\max_{pa}(\Upsilon_{pa:ap}(\Box))}(\\
&\qquad \Upsilon_{\mathcal{A}(e_3):e_3}(\Pi^D_{\mathcal{A}(e_4)}(e_4)))))) \\
&= \quad \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\\
&\qquad e_4\Gamma_{ct;\mathcal{A}(e_4)=A'_4;count\circ\Pi_{pb}}\Pi_{A'_4:\mathcal{A}(e_4)}(\sigma_{pp<\max_{pa}(\Upsilon_{pa:ap}(\Box))}(e_4 \times e_3)))) \\
&= \quad \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\\
&\qquad e_4\Gamma_{ct;\mathcal{A}(e_4)=A'_4;count\circ\Pi_{pb}}\Pi_{A'_4:\mathcal{A}(e_4)}(e_4 \bowtie_{pp<\max_{pa}(\Upsilon_{pa:ap}(\Box))} e_3)))
\end{aligned}
$$

Outer Join Instead of unnesting via a binary grouping operator (Eqv. 4.23), we can also apply a combination of outer join and unary grouping (Eqv. 4.24):

$$
\begin{aligned}
&\overset{(4.24)}{=} \quad \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(e_4 \bowtie^{ct:0}_{\mathcal{A}(e_4)=A'_4} (\\
&\qquad \Pi_{A'_4:\mathcal{A}(e_4)}(\Gamma_{ct;=\mathcal{A}(e_4);count\circ\Pi_{pb}}(\sigma_{pp<\max_{pa}(\Upsilon_{pa:ap}(\Box))}(\\
&\qquad \Upsilon_{\mathcal{A}(e_3):e_3}(\Pi^D_{\mathcal{A}(e_4)}(e_4)))))))) \\
&= \quad \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(e_4 \bowtie^{ct:0}_{\mathcal{A}(e_4)=A'_4} (\\
&\qquad \Pi_{A'_4:\mathcal{A}(e_4)}(\Gamma_{ct;=\mathcal{A}(e_4);count\circ\Pi_{pb}}(e_4 \bowtie_{pp<\max_{pa}(\Upsilon_{pa:ap}(\Box))} e_3)))))
\end{aligned}
$$

Evaluation The following table shows the results for the nested and both unnested versions of the query. Again, the evaluation of the unnested expressions is considerably faster than the evaluation of the nested one (with the binary grouping being [slightly] slower than the outer join).

Size	100	1000	10000
Nested	1.53s	132.65s	∞
Binary Grouping	0.15s	1.04s	64.93s
Outer Join	0.15s	0.94s	58.54s

Sequence-Valued Attributes in Both Expressions

We now come to the most complicated case, in which we allow sequence-valued attributes in both query blocks, the outer and the inner one. As an example query we take a modified version of the query presented in Section 4.5.5. For each book we determine how many books its authors have edited:

```
for $b in doc("bib.xml")//book
return
  <book-editor>
    { $b }
    <count> { count(for $c in doc("bib.xml")//book
                    where $b/author = $c/editor
                    return $c)
    } </count>
  </book-editor>
```

Normalizing this query introduces yet again several **let** clauses:

```
for $b in doc("bib.xml")//book
let $ba := $b/author
let $cc := (for $c in doc("bib.xml")//book
            let $ce := $c/editor
            where some $e in $ce
```

```
                    satisfies  some $ea in $ba
                                 satisfies  $e eq $ea
            return $c)
let $ct := count($cc)
let $ci  := <count> { $ct } </count>
let $sq  := ($b, $ci)
let $res := <book−editor> { $sq } </book−editor>
return $res
```

Normalization and translation into our algebra results in the following expression:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(b,ci)}(\chi_{ci:\mathcal{C}(\text{elem},s2,ct)}(\chi_{ct:count(cc)}(\chi_{cc:\Pi_c(\sigma_{\exists et\in\Upsilon_{e:ce}(\Box):\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}(e_2))}(e_1))))))$$

where

$$e_1 := \chi_{ba:b/author}(\Upsilon_{b:doc//book}(\Box))$$
$$e_2 := \chi_{ce:c/editor}(\Upsilon_{c:doc//book}(\Box))$$

and

```
doc := doc("bib.xml")
s1  := "book-editor"
s2  := "count"
```

Binary Grouping In a first step, we merge map operators and then unnest the nested implicit grouping expression:

$$\overset{(4.31)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_c(\sigma_{\exists et\in\Upsilon_{e:ce}(\Box):\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}(e_2)))}(e_1)))$$
$$\overset{(4.25)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}(\sigma_{\exists et\in\Upsilon_{e:ce}(\Box):\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))))$$

In a second step, we unnest the existentially quantified expressions introduced by the normalization and eliminate unnecessary unnest map operators:

$$\overset{(4.1)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}(\Pi^{tid_A}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}(\Upsilon_{e:\Upsilon_{e:ce}(\Box)}(tid_A(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))))))))$$
$$\overset{(4.1)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}(\Pi^{tid_A}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\Pi^{tid_B}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{e=ea}(\Upsilon_{ea:\Upsilon_{ea:ba}(\Box)}(tid_B(\Upsilon_{e:\Upsilon_{e:ce}(\Box)}(tid_A(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2))))))))))))$$
$$\overset{(4.32)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}(\Pi^{tid_A}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\Pi^{tid_B}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{e=ea}(\Upsilon_{ea:ba}(tid_B(\Upsilon_{e:ce}(tid_A(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2))))))))))$$

In a last step, we want to turn the cross product into a join operator. Before being able to do so, we have to eliminate one of the tid operators and push the other into the cross product. After having assigned the tid A, we unnest and then assign the tid B. This guarantees that for every value of B, we have the same value for A, so $B \rightarrow A$. Therefore, we do not need the duplicate elimination based on attribute B anymore (and can get rid of the operation to

assign it):

$$
\begin{aligned}
&\overset{(4.34)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}(\\
&\quad \Pi^{tid_A}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{e=ea}(\Upsilon_{ea:ba}(\Upsilon_{e:ce}(tid_A(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))))))))\\
&\overset{(4.33)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}(\\
&\quad \Pi^{tid_{A_1},tid_{A_2}}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{e=ea}(\Upsilon_{ea:ba}(\Upsilon_{e:ce}(tid_{A_1}(\Pi^D_{\mathcal{A}(e_1)}(e_1))\times tid_{A_2}(e_2)))))))))\\
&= \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}(\\
&\quad \Pi^{tid_{A_1},tid_{A_2}}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\Upsilon_{ea:ba}(tid_{A_1}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))\bowtie_{e=ea}\Upsilon_{e:ce}(tid_{A_2}(e_2)))))))
\end{aligned}
$$

Outer Join Instead of applying Eqv. 4.25 in the second rewrite of the first step above, we could use Eqv. 4.26 based on the outer join operator. After doing so, we can rewrite the existentially quantified subexpression as shown above:

$$
\begin{aligned}
&\overset{(4.26)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\ \bowtie^{ct:0}_{\mathcal{A}(e_1)=A_1'}(\Pi_{A_1':\mathcal{A}(e_1)}(\Gamma_{ct;=\mathcal{A}(e_1);count\circ\Pi_c}(\\
&\quad \sigma_{\exists et\in\Upsilon_{e:ce}(\square):\exists eat\in\Upsilon_{ea:ba}(\square):e=ea}(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2))))))\\
&= \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\ \bowtie^{ct:0}_{\mathcal{A}(e_1)=A_1'}(\Pi_{A_1':\mathcal{A}(e_1)}(\Gamma_{ct;=\mathcal{A}(e_1);count\circ\Pi_c}(\\
&\quad \Pi^{tid_{A_1},tid_{A_2}}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\Upsilon_{ea:ba}(tid_{A_1}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))\bowtie_{e=ea}\Upsilon_{e:ce}(tid_{A_2}(e_2))))))))
\end{aligned}
$$

Evaluation The following table summarizes the results for the running times of the different versions of the query. This query does not seem to be favorable to unnesting. However, we can exploit the fact that the aggregate function *count* is insensitive to order. Hence, we can employ efficient implementations for the equijoin and the unary grouping operator. This results in substantially more efficient plans with notable advantages for the plan using binary grouping.

Size	100	1000	10000
Nested	0.84s	67.96s	∞
Binary Grouping	0.14s	0.84s	9.57s
Outer Join	0.14s	1.04s	35.99s

4.6. Implementation

To validate the feasibility of our unnesting framework, we have implemented most equivalences presented in this chapter (see [Bit07] for details). In this section, we present the basic design that underlies our implementation. We also discuss the performance of our rewriting component.

4.6.1. Rules

Let us first discuss how we get from equivalences presented in the previous sections to rules and why we need to distinguish both concepts. After that, we look at the implementation of rewrite rules in Natix.

From Equivalences to Rewrite Rules

Equivalences are valid when applied in both directions. However, in the case of unnesting equivalences we prefer the unnested representation of a query to the nested one. Consequently, we apply the unnesting equivalences introduced in the previous sections from left to right.

This view of directed application of equivalences is called *rewrite*. Hence, we will implement all unnesting equivalences as rewrite rules and embed them in a rewriting system that selects the most specific rewrite rule to apply.

A rewrite rule consists of three parts: (1) It matches a pattern in the query. (2) It tests conditions that must hold in addition to the structural properties expressed in the query pattern. (3) It restructures the query based on the variable bindings established during matching.

As an example consider Eqv. 4.3 which holds only under the conditions C discussed in Sec. 4.3

$$\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) \quad = \quad e_1 \ltimes_{A_1=A_2 \wedge p} e_2.$$

We denote this equivalence as a directed rewrite

$$\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) \quad \overset{C}{\Rightarrow} \quad e_1 \ltimes_{A_1=A_2 \wedge p} e_2.$$

The left-hand side of this rewrite indicates the pattern to match. We write the additional conditions C to verify before the rule is applied above the arrow. The right-hand side specifies the result after application of the rewrite rule. Obviously, we are very specific about the structure of certain parts of the pattern to match while we are more tolerant for other parts of the pattern. For the former, we precisely state the pattern to match, e.g. the two selections, the existential quantifier, and the comparison operator $=$ in the correlation predicate and their structural relationship. For the latter, we use typed pattern variables, in this example e_1, e_2, A_1, A_2, x, and p. They denote parts of the pattern that need to match with algebraic operators or expressions in the query that are consistent with the types required by the pattern. The condition C narrow the possible matches for these variables. In our example, the condition includes that e_1 and e_2 can be evaluated independently.

Besides equivalences for which we clearly prefer one direction of application, there are other rewrites where we cannot state such a preference. For example the support rewrites can be beneficial when applied in either direction. Thus, we create one rewrite to implement either direction to support these rewrites in a rule-based rewriting system. As a result, we need to take care that we do not run into cycles when we apply support rewrites. This problem can be resolved by memorizing all expressions we have generated so far during rewriting [GD87, McK93]. When we create a new expression, we first check, if the memo table already contains an isomorphic query pattern. If not, we add the new expression to the memo table and resume rewriting. Notice that our task is easier than memorizing all possible plan alternatives because after one unnesting step we can be certain that only structurally different query plans will be generated. Thus, after each successful application of an unnesting equivalence, we can discard all entries in the memo table. This is different to the exhaustive search performed in [GD87, McK93].

Rule Implementation

From the previous discussion follows that we have to solve two problems when we implement a unnesting rewrite [PHH92]. (1) We have to test if a subexpression in the plan matches the pattern and passes all conditions. We will refer to this task as *rule matching*. (2) Once we have matched the pattern, we have bindings for the variable parts in the algebraic pattern. Given these bindings, we can now construct a new algebraic expression which yields the result of the rewrite. We will refer to this task as *rule application*.

We have implemented the rewrite rules with mutators. Mutators differ from visitors [GHJV95] because their visit function might modify the object structure during the traversal over the query graph. Thus, when we try to apply some rewrite implemented by a mutator, we traverse the query. While descending in the depth-first traversal, we can gather information needed for examining subexpressions. While ascending, we first match each operator in the query to the pattern and condition implemented in the rewrite. When rule matching

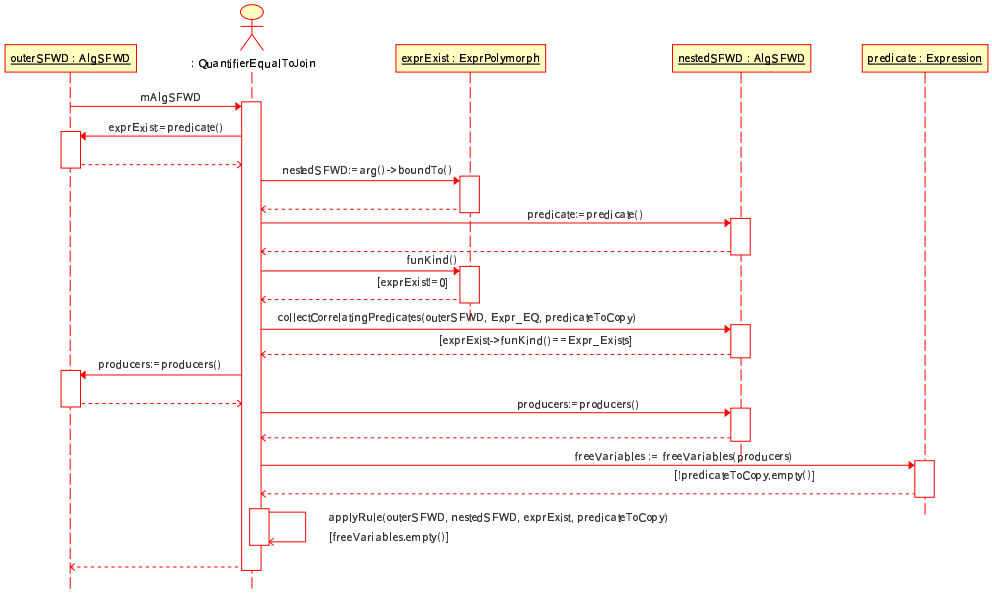


Figure 4.11.: Rule Matching in the Mutator for Eqvs. 4.3 and 4.15

succeeds, we apply the rewrite to the subexpression. The possibly rewritten subexpression is returned as a result of each recursive call to the visit function.

In our implementation, we merge two algebraic equivalences into a single mutator when they match almost the same algebraic pattern. Thereby, we reduce the number of traversals over the query and the number of matching operations. At the same time we keep the implementation easy to comprehend and flexible to apply. In particular, as we will see shortly, we can schedule rewrites effectively.

Since the basic ideas of the implementation is the same for all our unnesting equivalences, we will use Eqv. 4.3 as an example. Notice that in this equivalence, we only need to replace the existential quantifier by a universal quantifier to arrive at Eqv. 4.15. When we look at the decision tree in Fig. 4.14, we observe that the path to either equivalence is the same. This is the main reason for implementing both equivalences in one single mutator. But in our subsequent discussion we will concentrate on the implementation of Eqv. 4.3. In Fig. 4.11 we trace the execution of the mutator assuming Eqv. 4.3 can be applied.

Rule Matching Let us start with the modification function `mAlgSFWD` (the modification function corresponds to the visit function for visitors [GHJV95]). Its main purpose is to check if either of the two equivalences matches. The mutator assumes that during normalization and translation quantifiers are moved into the predicate of the SFWD block. Universal quantifiers are translated into negated existential quantifiers. Fig. 4.12 depicts the internal representation of a nested SQL query after translation. In the first step, we look for an existential quantifier in the predicate. If we find one, it is bound to the variable *exprExist*.

In the next step we look for an immediately nested SFWD block and bind it to the variable *nestedSFWD*. For simplicity we ignore aggregation or grouping here. We use the function `collectCorrelatingPredicate` to search for a correlation predicate in the `AlgSFWD` block referenced by variable *nestedSFWD*. Since we only allow equality predicates, we pass the constant `ExprEQ` to this function. At the end, the variable *predicateToCopy* stores all comparison functions (instances of class `ExprFOpCall`) which correlate the inner query block with the outer one.

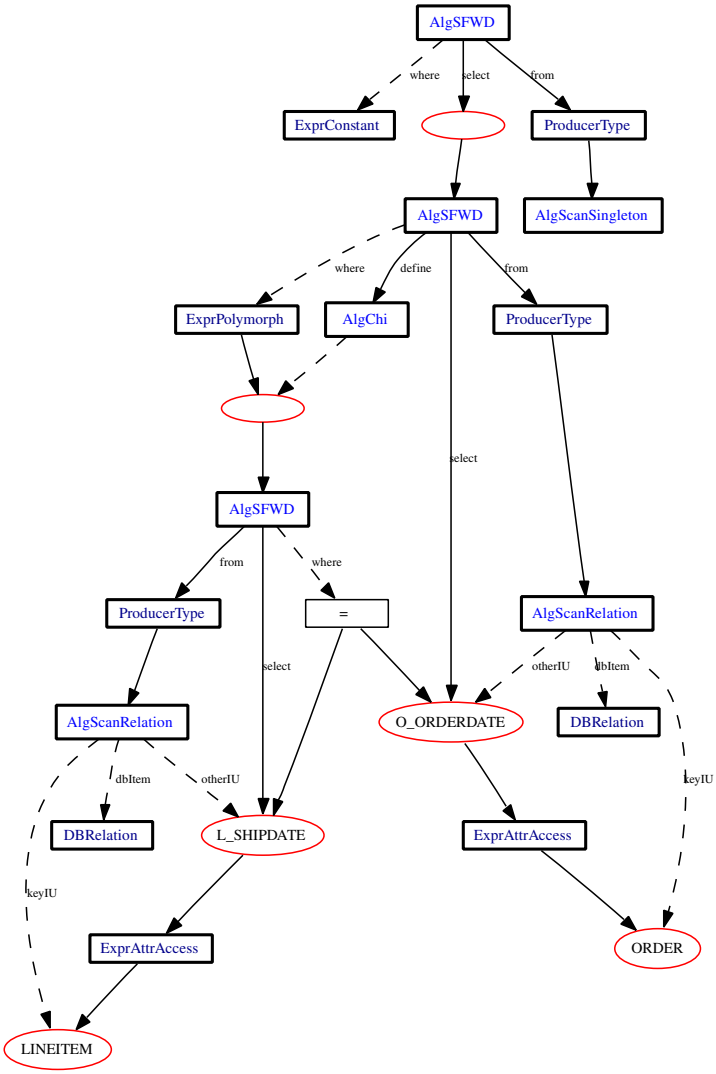


Figure 4.12.: Nested query with existential quantifier and correlation predicate =

It remains to verify that the nested query block can be evaluated independent of the outer query block. Therefore, we check if the nested query block binds all variables it refers to. To obtain all free variables in *nestedSFWD*, the mutator calls the function `freeVariables`. If at least one correlation predicate exists, and the check for free attributes has failed, function `mAlgSFWD` calls function `applyRule`.

Since universal quantifiers are translated into negated existential quantifiers, rule matching proceeds similar to the existential case.

Rule Application Based on the variable bindings established during rule matching, function `applyRule` restructures the query rooted at the `AlgSFWD` block bound to variable *outerSFWD* into an unnested one.

First, it creates an instance of the class `AlgJoin` and annotates it as left semijoin. The expression on the left-hand side of the semijoin operator is represented by the `AlgSFWD` block bound to variable *outerSFWD*. The expression bound to variable *nestedSFWD* becomes the right argument of the semijoin. The correlation predicate bound to variable

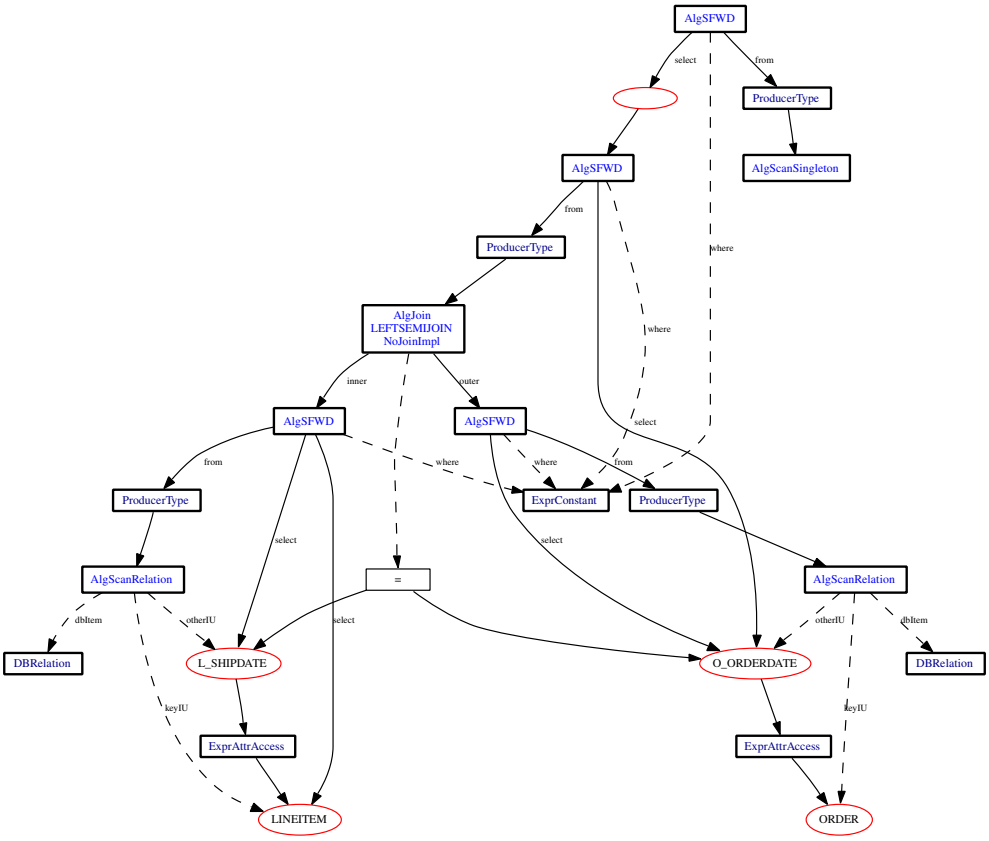


Figure 4.13.: Result of Unnesting with Eqv. 4.3

predicateToCopy becomes the predicate of the new semijoin. The projection list of this semijoin corresponds to the projection list of the old outer `AlgSFWD` block.

Having created the semijoin, we need to remove the quantifier from the predicate of the outer `AlgSFWD` block referenced by variable *outerSFWD*. We also have to delete the correlation predicate in the predicate of the nested `SFWD` block referenced by variable *nestedSFWD*. Now function `applyRule` removes the instance of class `AlgChi` from the outer `AlgSFWD` block, which materializes the inner expression from the define list.

Finally, we create a new instance of class `AlgSFWD` which wraps the resulting expression. This is necessary because the cost-based optimizer is triggered for each `AlgSWFD` block. The only producer of this block becomes the semijoin created in the first step. Then, we set the predicate of this block to true and copy the projection list of the join into the projection list of this block. Eventually, the function `applyRule` returns this wrapper `AlgSFWD`. The resulting plan representation is shown in Fig. 4.13.

4.6.2. Rule Scheduling

An effective rewriting engine must choose the rewrite rule that results in the most efficient plan. We call this task *rule scheduling*. The decision trees presented in the previous sections guide this selection. For convenience, we repeat them in Figs. 4.14 and 4.15.

Let us recapitulate, how we select the most specific equivalence given an algebraic pattern: We enter at the root of any of the two decision trees and check for the basic pattern. If it matches we traverse the decision tree top-down. At each inner node of the tree we test if a certain condition holds. Based on the outcome of the test, we resume the traversal until

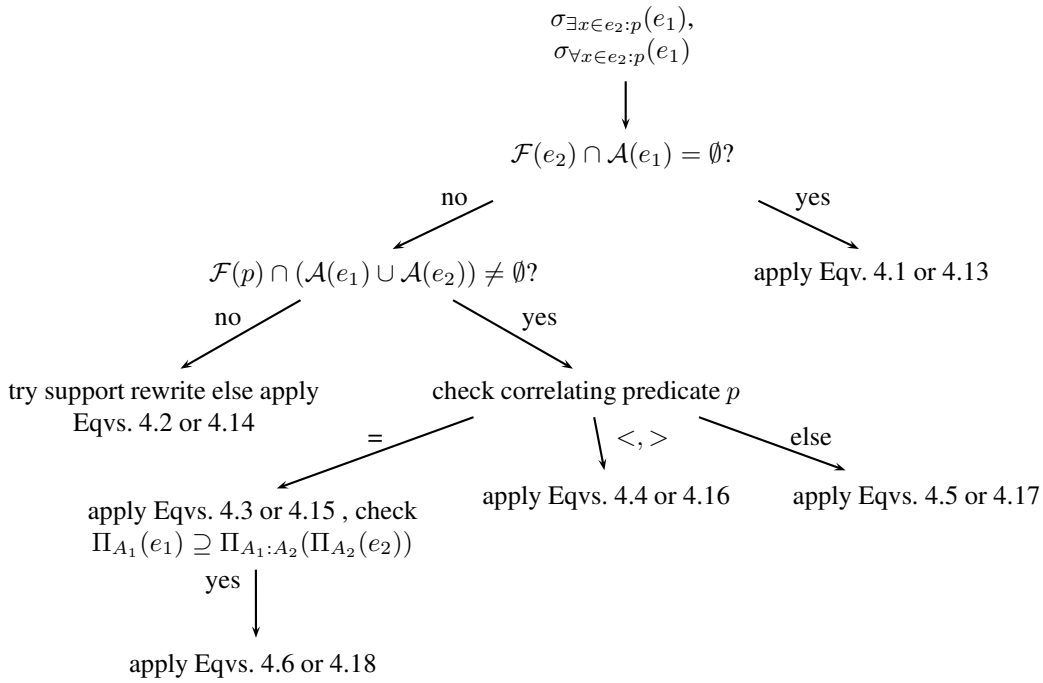


Figure 4.14.: Combined Decision tree for quantified queries

we reach a leaf node. Each leaf node tells the equivalence to apply.

In our implementation, we reverse this logic and start at the bottom with the most specific rewrite rules. Each mutator that implements such a rule has to test all conditions on the path from the root of its decision tree to the leaf node. By scheduling the most specific rule first and the most general rule last, we make sure that an efficient algebraic expression is created for which an efficient plan can be generated during cost-based optimization. In principle, we could always apply the least specific rule, i.e. one that results in a d-join. But this requires further rewrites and complex pattern matching to achieve the same effect.

However, we do not need such a strict ordering among the rules: When two rules match disjoint patterns, i.e. not both of them can match at the same time, we group them into a *rule set*. In principle, all rules in a rule set can be tested in an arbitrary order. For example, for the decision tree in Fig. 4.14, we can test Eqvs. 4.3 and 4.4 at the same time. The reason is that we always match the least specific type of comparison in the correlation predicate and this cannot be both an equality and inequality predicate. We plan to improve this simple test by spliting conjunctive predicates and matchin the most specific part as we have outlined in [MM05b].

This line of reasoning leads us to the rule sets shown in Fig. 4.16. Consider rule set 0. It contains all unnesting equivalences that result in a unary grouping operator. Since all these equivalences share a common path to the root of the decision tree with another, more general unnesting equivalence, we have to put these rewrites into a separate rule set. These more general equivalences are contained in rule set 1 (Eqvs. 4.3, 4.15, and 4.29). They are used when the condition $\Pi_{A_1}(e_1) \supseteq \Pi_{A_1:A_2}(\Pi_{A_2}(e_2))$ ($e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$ resp.) does not hold. As mentioned above, we can also include the unnesting equivalences for quantified queries that match for correlation predicates containing $<$, $>$, $\leq$ or $\geq$. Next, in rule set 2, we put all unnesting equivalences that allow an arbitrary correlation predicate. These rules need to be in a separate rule set because they will also match for all expressions that should be handled by the rewrites in rule set 0 and rule set 1. We now turn our attention to the support rewrites contained in rule set 3. Remember that we try the rewrite rules before

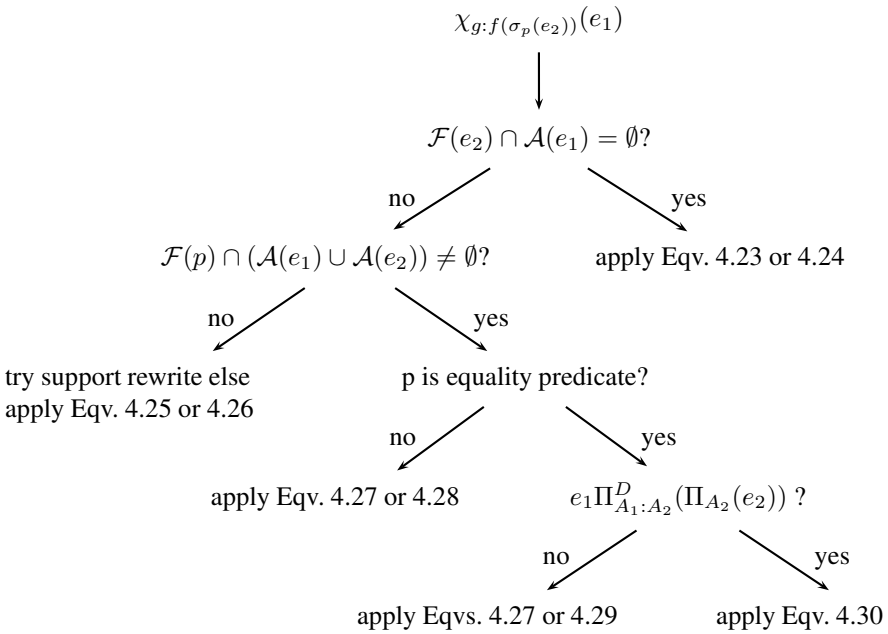


Figure 4.15.: Decision tree for implicit grouping

we apply equivalences that might result in cross products or d-joins. Hence, we must test all support rewrites before we check the latter equivalences. Finally, since we prefer cross products to d-joins, we conceptually create two rule sets. The first consists of the unnesting equivalences that introduce cross products and the latter (rule set 5) contains the remaining unnesting equivalences that introduce d-joins.

Thanks to Eqvs. 2.17 and 2.18, we are even able to generalize our implementation: We can remove all equivalences contained in rule set 4 and generate unnest map operators first. In the next step, we can replace the unnest map operator by a cross product if the subscript of the unnest map operator can be evaluated independent of the argument of the unnest map operator. This sequence of rewrites generates the same algebraic expressions as the removed rewrites did, but our rules become more generally applicable.

In Fig. 4.17, we present the pseudo code for rule scheduling. After initializing the rule sets, we first apply the three most specific rule sets in the fixed order. The rules contained in each rule set are triggered in function `tryRuleSet` or `trySupport`. In our current implementation, we iterate over all rules contained in the rule set and try to apply each rule. This allows us to merge adjacent rule sets when they both contain unnesting or support rewrites. In our case, we can merge the first three rule sets and the last two rule sets.

Rule set	Equivalences
0	4.6, 4.18, 4.30
1	4.3, 4.4, 4.15, 4.16, 4.29
2	4.5, 4.17, 4.27, 4.28
3	all support rewrites
4	4.2, 4.14, 4.25, 4.26
5	4.1, 4.13, 4.23, 4.24

Figure 4.16.: Rule sets for unnesting equivalences

```
1  AlgSFWD* unnest(AlgSFWD* expression){
2
3     RuleSet[6]  decisionTree ;
4     for  (i = 0  to  5)  {
5        decisionTree [i ]  =  createRuleSet (i );
6     }
7     bool  unnested  = true;
8
9     while(unnested){
10       unnested  =  false ;
11
12       for  (i = 0  to  2)  {
13          unnested  =  tryRuleset (i ,  expression );
14       }
15       /* try to apply preparing support rewrite */
16       if (unnested)  continue;
17
18       unnested  =  trySupport (3,  expression );
19       if (unnested)  continue;
20
21       for  (i = 4  to  5)  {
22          unnested  =  tryRuleset (i ,  expression );
23       }
24    }
25    return  expression ;
26 }

27 bool  tryRuleset (UnnestMutator[]  ruleset ,  AlgSFWD*\& expression){
28       foreach (mutator  in  ruleset ){
29          expression .acceptM(mutator);
30          bool  changed = mutator.unnested ();
31          if (changed)  break; /* unnesting rewrite successful */
32       }
33       return  changed;
34 }

35 bool  trySupport (SupportMutator[]  ruleset ,  AlgSFWD*\& expression){
36    foreach (mutator  in  ruleset ){
37       mutator. reset ();
38       if (mutator  not  applied  here  before  in  the  other  direction ){
39          expression .acceptM(mutator);
40          bool  changed = mutator.unnested ();
41          if (changed)  break; /* support rewrite successful */
42       }
43    }
44    return  changed;
45 }
```

Figure 4.17.: Pseudo code for rule scheduling

Notice that we could call different functions that trigger the contained rules and thereby support arbitrary triggering strategies in a controlled manner. Thus, our approach is similar to the one proposed in [PLH97].

Termination of Rewriting

From the pseudo code in Fig. 4.17 it is also clear that rewriting will eventually terminate. If no rewrite was applied in one iteration of the main loop in function `unnest`, unnesting terminates. Furthermore, every unnesting rewrite removes a nested expression in some subexpression of the query. It does not introduce new nested expressions. Consequently, at most n applications of unnesting rewrites will occur, if the query contains n nested query blocks. Finally, support rewrites might be applied if none of the unnesting rewrites of the first three rule sets matches. The support rewrites in our framework can be applied in both directions. Thus, duplicate algebraic expressions might be constructed leading to loops in the application of support rewrites. As discussed in Sec. 4.6.1, we avoid this problem by memorizing all expressions created by support rewrites. Hence, we can detect and discard those duplicates and thereby guarantee termination.

Summarizing our results so far, our rule-based rewriting component is effective because it always applies the most specific rewrite possible as prescribed by the decision trees presented in Sections 4.3, 4.4, and 4.5. In most cases, we implement two equivalences that match very similar patterns in one mutator. Thereby our implementation remains comprehensive and easy to extend and tune. In the other hand, our approach requires matching similar patterns multiple times and one traversal for each unnesting mutator. In the remainder of this section, we evaluate the performance impact of these design decisions.

4.6.3. Evaluation

In this section we assess the efficiency of our unnesting component. We will investigate two aspects: (1) What is the overhead of rule application when unnesting rules can be applied? (2) What is the overhead for queries that do not contain nested query blocks. For the former question, we are willing to invest time because we can expect improved query performance in orders of magnitudes, i.e. queries finish within seconds instead of running for several hours. For the latter problem, we want to minimize the effort spent to discover that unnesting is not necessary.

For our experiments, we run Natix on a Linux Server with 4 Intel(R) Xeon(TM) CPUs (3.40 ,Ghz, 2 MB), a 3 GB hard disk and SuSE Linux 10.0 as the operating system. Natix was compiled with GCC 4.0.3 and optimization level O2.

In Fig. 4.18, we present the elapsed time for parsing, translation and unnesting. For each query we give the average elapsed time in milliseconds of 1000 executions. We compare the time for all three steps with all unnesting equivalences turned on (with unnesting) and turned off (w/o unnesting). The labels on the x-axis tell applicable unnesting equivalence. The first three queries are taken from the TPC-H benchmark (Query 2, 5, 9).

The plot in Fig. 4.18 shows that applying unnesting equivalences causes an overhead of at most 5 milliseconds. This is a very satisfying result because we can expect improved query execution times that easily match this additional effort.

On the other hand, the TPC-H query 5 and 9 are reasonably complex queries without nested query blocks. For these two queries, we almost do not observe any overhead. More detailed experiments presented in [Bit07] did not reveal any particularly expensive unnesting equivalence.

4.7. Summary

Unnesting Equivalences Our unnesting equivalences detect and unnest a wide range of nested queries. For quantified queries, we can even unnest queries with query blocks whose correlation predicates span multiple blocks. Unfortunately, this is not the case for queries containing implicit grouping. Here, we advocate magic decorrelation [SPL96] to

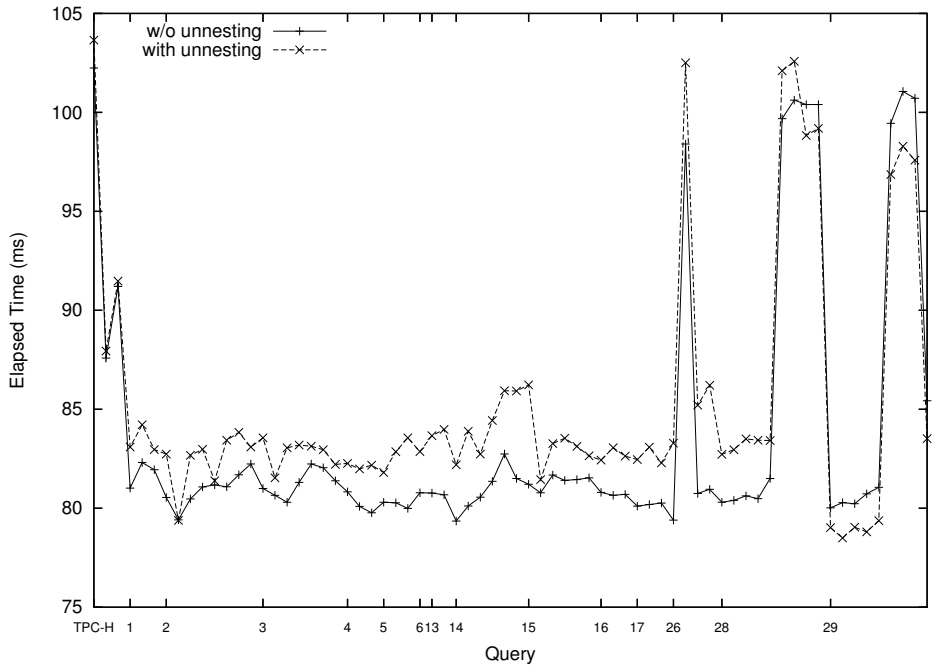


Figure 4.18.: Overhead of query unnesting

efficiently treat such nested queries. Notice that currently it is unknown how this technique needs to be applied to preserve order.

In our treatment we have restricted ourselfs to conjunctive predicates. In conjunctive predicates containing several conjuncts that are correlation predicates, our framework always choses the rewrite that is consistent with the least specific correlation predicate contained in any conjunct, i.e. in $A_1 = A_2 \wedge B_1 \neq B_2$ the equivalence for arbitrary θ-predicates are applied. In some cases, we can do better. For binary grouping and thus implicit grouping, we have discussed more efficient alternatives [MM05b]. The idea is to look for the conjuncts that lead to the most specific rewrite and treat the remaining conjuncts as residual predicates. We believe that a similar treatment is also possible for semijoins and antijoins.

In several cases, more efficient query execution plans can be derived when the correlation predicate contains disjunctions. Some proposals for treating disjunctive predicates [BMM06, BMM07, EGLGJ07] will lead to more efficient query execution plans. Thus, we need to integrate these ideas into our framework.

Rule Scheduling Our experiments show that our framework leads to an effective and efficient implementation. But the approach to rule scheduling presented here leaves many opportunities to tune rewriting. As one possible extension we can keep global information about nested query blocks, quantifiers, or free variables to prune application of certain rules. These information can be gathered by a single traversal over the query graph or as a side effect during rule matching. We can even go a step further and not only record the existence of certain features but also provide direct access to, e.g. nested query blocks or free variables. As another possible improvement, we could share information about partially matched query patterns among mutators and thereby save matching effort. However currently, as we have seen in the experiments in Section 4.6.3, we do not have to. The time spent for rule matching and repeated traversals over the query is neglegible compared to the overall optimization time. Consequently, we expect our architecture to scale well even when we add many more rules to our rule-based rewriting engine.

Complementary Rewrites As already pointed out, our architecture allows us to plug further rewrites into our rewriting component. It is natural to complement our unnesting equivalences with further rewrites which we will briefly discuss here.

First, a weak spot in our unnesting approach are queries containing implicit grouping in which correlation predicates span multiple query blocks. Our unnesting equivalences cannot detect such patterns and hence, will unnest such queries using d-joins. As this might result in unsatisfactory query execution times, we want to implement magic set optimizations for such cases [MFPR90, SPL96].

Second, we point out that the quantified queries are insensitive to order and duplicates. Consequently, it makes sense to include rewrites that propagate these information in the query as proposed by Pirahesh [PLH97] and Mumick [MP94]. As a result, our cost-based query optimizer has more freedom in chosing operator implementations and to (re-) order the processed data. In the context of XQuery, this is even more important because by default the result of XPath expressions must be returned in document order. Furthermore, the semantics of **for** clauses is defined in terms of sequence order [BCF+07]. The optimizations proposed by Grust et. al [GRT07] can be used as a starting point here.

Third, our unnesting equivalences that detect implicit grouping introduce outerjoins and grouping operators. The proper placement of outerjoins and grouping can have enormous performance impact. Hence, we plan to integrate the optimizations developed in [RGL90, GLR97] for outerjoins and in [CS94, YL94] for grouping into our rewriting component.

Our results show that in some cases it is either not possible or not beneficial to unnest a query. Consequently, query unnesting should be integrated into the cost-based query optimizer. Guravannavar [GRS05] present cost-based optimizations in the presence of nested queries. The framework proposed in [EGLGJ07] discusses alternative processing strategies for nested queries and their trade-offs.

4.8. Related Work

Processing Nested Queries The problem of efficient processing of nested queries first occurred for SQL. The original technique proposed was to evaluate the inner query block for each tuple of the outer block [AC75].

Graefe showed that this straightforward evaluation of nested queries loops can be improved by several techniques [Gra03]. As a consequence, query unnesting should be integrated into the cost-based query optimizer [SHP+96, GRS05, ALW+06, EGLGJ07]. Nevertheless, unnesting nested queries leaves more freedom for subsequent algebraic optimizations possibly reintroducing nested queries with efficient execution strategies.

Unnesting in SQL Kim was the first to observe that it is possible to rewrite a nested SQL query into an unnested one and thereby significantly improve the evaluation cost [Kim82]. He introduced a classification for nested queries and pointed out that nested queries can be unnested such that the transformed query uses joins or grouping instead of nested queries. However, restrictions required for their validity have been found for some of his rewrites. They mainly concern empty results for the inner query block, NULL values, and duplicate handling.

Algebraic Approaches to Unnesting Several solutions for these problems were proposed. Current approaches differ from early approaches to query unnesting because unnesting of queries either works on algebraic [Mur89, Mur92, CM93, CM95b, Ste95, SABdB94, GLJ01] or calculus representations [Nak90, Feg98, FM00, SPL96] of the query. A major advantage of unnesting at the algebraic level is that now unnesting can be integrated into cost-based plan generation [GLJ01].

Several rewrites introduced grouping, outerjoins, and semijoins which increased the expressiveness of SQL and widened the range for additional optimizations.One of the most

important constructs to avoid problems with NULL-values and empty results when unnesting queries turned out to be outer joins [Kie84, Day87, GW87]. After their introduction into SQL and their usage for unnesting, reordering of outer joins became an important topic [BGI95, GLR97, RGL90].

Other rewrites detect division operators [RM06] or discuss efficient implementations for queries containing universal quantifiers [CKMP97]. Claussen et al. [CKMP97] compare evaluation techniques for universal quantifiers. Implementation techniques for relational division are presented in [GC95, RSMW02]. In [RM06] algebraic rewrites for the division operators are presented.

Unfortunately many proposals to unnesting are restricted to sets. But since most SQL queries have bag semantics, proper treatment of duplicates is important [Klu82, PHH92, SPL96].

Recently, nested queries containing disjunctions received attention [BMM06, BMM07]. Before that, disjunctions were treated by duplicating subexpressions and introducing union operators. By resorting to bypass-operators [CKM$^+$00] it is possible to construct query execution plans that outperform previous techniques by orders of magnitudes.

Unnesting in XQuery Optimization of XQuery can benefit from the techniques mentioned so far for SQL queries that do not need to preserve order or when order is explicitly treated in an unordered query processing environment. The latter can be achieved by translating XQuery into SQL [GST04] or into a relational algebra [PCS$^+$05, LKA05], unnesting the query, and adding a final sort. While this technique is feasible, we argue in [MHKM04] that the decision to destroy and later repair document order should be based on costs. One contribution of this work is to point out when no sorting is needed after unnesting nested queries in an order-preserving query processor.

XQuery lacks an explicit grouping construct — a situation that is likely to be remedied [BC04, BCC$^+$04, BCC$^+$05, Eng07]. Until then, grouping must be formulated implicitly, giving rise to another stereotype of nested queries. But even when explicit grouping arrives in XQuery, nested queries will probably still be used sometimes to express grouping implicitly. Detecting and unnesting implicit grouping is a challenging task, before us Paparizos et al. tried to tackle it [PAKJ$^+$02]. In their approach a tree pattern based grouping operator is proposed, and a single case where it can be beneficially used to unnest a nested query is identified. However, the description is at a rather high level and special cases are not taken care of, e.g. empty groups. Based on their previous work [Feg98, FM00], Fegaras tried to adopt his approach to XQuery [FLBC02]. However, from his exposition it is not clear whether the unnesting techniques presented there preserve order. [DPX04] present an algorithm for detecting grouping on a subset of XQuery. Their algorithm minimizes the number of navigation steps needed to evaluate a query. However, their algorithm does not preserve order semantics as required in XQuery. For XQuery queries which cannot be unnested, the evaluation techniques proposed by Sartiani [Sar03a] can by applied. Recently, an algebraic unnesting framework similar to ours was proposed [RSF06]. To the best of our knowledge, it is the only other treatment of nested queries that fully obeys to XQuery semantics.

Some of the material presented here has already appeared elsewhere [MHM03a, MHM03c, MHM03b, MHM04, MHM06]. We extend this work by discussing further evaluation strategies for unnested query execution plans. Furthermore, we present the implementation of our unnesting equivalences. We also investigate the effort in terms of optimization time needed to unnest nested queries.

Closely connected to the efficient evaluation of XQuery is that of XPath [GKP02, GKP03b, BKHM05]. Since XPath expressions are translated into our algebra, our unnesting techniques can also be applied to them.

Rule-based Optimization Currently query unnesting is applied in a rewrite phase and thus precedes the cost-based plan generation. The main reason for this decision is that in the overwhelming number of cases the unnested query is at least as fast to evaluate as the nested one. But compared to the nested query, cost-based optimization has more opportunities to optimize the unnested query because query execution plans are constructed per query block. This restricts the information available to the plan generator to smaller fragments of the query and also restricts the possibilities to reorder operators in the query execution plan.

However, a major advantage of unnesting at the algebraic level is that unnesting can be integrated into cost-based plan generation [SHP+96, GLJ01, ALW+06]. Only during cost-based plan generation it is possible to decide wether unnesting is beneficial for unnesting techniques which only sometimes improve performance. For example when we duplicate subexpressions the resulting query execution plan can be less efficient to evaluate.

In any case, unnesting equivalences will be embedded into rules, and these rules will be integrated into a rule-based query optimizer. Rule-based optimization is the major implementation technique for both heuristic rewriting [HFLP89, PHH92, PLH97, CZ96, CZ98, Che98] and cost-based optimizers [CDF+86, GD87, GM93, GCD+94, Gra95, Fre87, Loh88] because it allows extensible implementations that are resonably efficient. However there is a tradeoff when we aim for comprehensible and provably correct implementations. On the one hand, several proposals exist to specify rules in a declarative way [FMS93, DB95, Che98]. These declarative rule specifications may also serve as input to a theorem prover that can prove, e.g. the correctness of the rules. On the other hand, coding rules directly [PHH92, PLH97, KD99] allows for optimizations that are not possible otherwise. The architecture of our rule-based rewriting component is closest to the one described in [PLH97].

5. Cost-Based Optimization

The task of cost-based optimization is to schedule all logical operators contained in the query and to choose the most efficient implementation for them. Usually, the optimizer is invoked per query block, in our case per `AlgSFWD` block. Heuristic optimizations, such as merging query blocks during query unnesting or view merging, are important preparation steps for cost-based optimization because the cost-based optimizer get a holistic view on the query to optimize and, thus, is able to generate better query execution plans.

Enumerating all possible operator orders and implementations efficiently is a challenging task, i.e. in general, finding the optimal join order is NP hard. For SQL and relational databases, it was shown that only by considering the full search space of plans, we can assure to find the optimal plan. We conjecture that this also holds for XQuery.

Previous work on cost-based query optimization assumed a data model based on set or bag semantics. Now, in XQuery, the order of the items in a sequence is the new aspect to consider. Unfortunately, joins in our algebra over sequences are not commutative (but still associative). Additionally, many other operators cannot be reordered, as we are accustomed to in the relational algebra over sets or bags. Thus, query optimizers are severely constrained in considering plan alternatives.

In this chapter, we show that the results developed in Chapter 2 provide important information, when reordering operators implies that order is destroyed. In particular, we have a choice: either we destroy order and pay the additional cost of sorting to repair it later, or we preserve document order and sequence order.

Our argument is based on experiments carried out with concrete queries and a benchmark data. In Section 5.1, we describe the structure of our data set which allows us to investigate the performance impact of different query parameters at fine granularity.

In the first set of experiments, presented in Section 5.2, we look at the issue of document order. Since XPath and XQuery demand the result of path expressions to be in document order, we are forced to return the result nodes of a path expression in document order. Consequently, we take care that every location step returns a node sequence in document order and duplicate free, or we repair them when we have to. In particular, we compare the evaluation of path expressions using navigation with indices. Using indices allows us to reorder the evaluation of axis steps. As pointed out above, evaluating location steps in a different order necessitates in sort operations to repair document order.

In the second set of experiments, discussed in Section 5.3, we turn our attention to sequence order. Sequence order is relevant when we combine the results of different sequences in a series of **for** clauses. As we have shown in Chapter 2, joins on sequences are associative but not commutative. Because exchanging the arguments of a join also changes the order of the result, we need sort operations to repair order. We show that there are situations where this additional cost is more than outweighed by the effort saved during join processing.

Our observations motivate the need for a cost-based optimization of XQuery. Thus, we present the architecture of our cost-based optimizer in Section 5.4. Clearly, an efficient support for properties, in particular of order information, is crucial for an efficient query optimization. Hence, we give details how our cost-based plan generator manages properties in a generic fashion. Fortunately, we can build upon efficient techniques for including order in query optimization [NM04].

5.1. The Benchmarking Data

To get precise performance characteristics for each of the evaluation strategies we discuss in this chapter, we use generated data sets. This allows us to tune the selectivity of XPath axis steps or join predicates individually.

The input documents used in our experiments were generated by the XDG document generator implemented by our group.[1] It allows to specify several parameters, i.e. the number of nodes, the document depth, the fan-out of each element and the number of different element and attribute names.

Conceptually, the generator creates as many child nodes as defined by the parameter "Fan-out" and resumes with a recursive call for each child. When the depth of the recursive calls reaches the specified parameter value "Depth", no recursive calls are executed any more. The frequency of occurrences of tag names decreases by a factor 2 for each subsequent tag name. E.g. the argument "C" for parameter "Elements" means that the tag names `A`, `B`, and `C` are used in the document where every second node gets tag name `A`, every fourth node gets tag name `B`, and so on. To get up to 100%, nodes with tag name `A` are generated. In our setup, this means that exactly 50.1% of the nodes are `A` nodes. The tool generates new nodes until the limit for the number of nodes (#Nodes) is reached.

In principle, this generator might introduce correlations between predicates such that the distribution of tag names strongly depends on the parent nodes. For our data sets, this is not the case for tag names `A`, `B`, `C`, and `D`. However, the remaining tag names only occur as leaf nodes.

In document D0, every element contains the attributes `a`, `b`, `c`, `d` and an `id`. The range of these attributes values doubles for each subsequent letter. This means that attribute `a` only takes the values 0 or 1, while attribute `c` may have the values from 0 to 7, the values of attribute `d` range from 0 to 15, while `id` takes a unique value for each element. The values for each attribute `a`, `b`, `c`, and `d` are uniformly distributed.

We generated documents of five sizes with the parameters summarized in Figure 5.1. In this figure, we give the size of the generated XML file. This setup allows us to control the selectivity of each axis step between 50% and 0.1% by changing the name test of each axis step. Similarly, we can change the selectivity of join predicates by combining different attribute names.

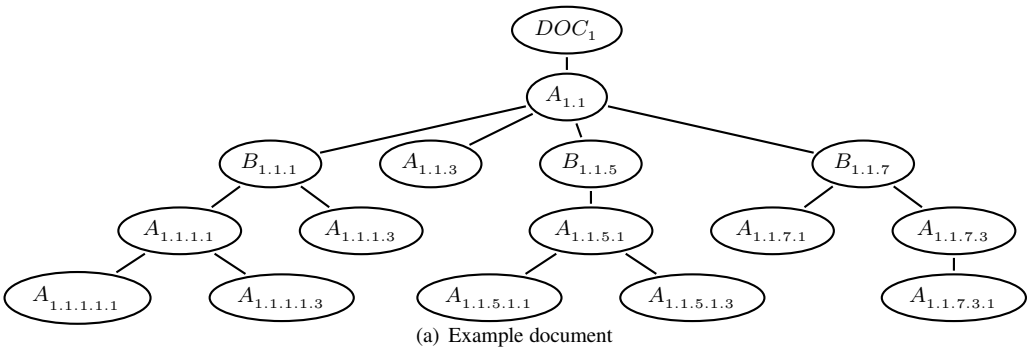
Document Parameter		Document Instance				
Parameter	Description	D0	D1	D2	D3	D4
#Nodes	# of generated XML elements	5,000	10,000	100,000	1,000,000	10,000,000
Depth	max. depth of the document	4	4	5	6	7
Fan-out	# of children per element node	10				
Elements	# of different tag names	"K" (11)	"J" (10)			
Attributes	# of different attribute names	4	0			
Size	Size of textual XML file	0.396MB	0.327MB	3.46MB	36.5MB	384MB

Figure 5.1.: Parameters and characteristics of generated documents

5.2. Document Order Considered Harmful

In this section, we investigate different query execution plans (QEPs) that either employ navigation or indices to evaluate path expressions. In a series of experiments, we show that sometimes it is more efficient to reorder location steps and sort the result of a path expression than to preserve document order even for intermediate results of path expressions. Moreover, we will see that there is no clear winner when we have to decide if navigation or

[1] available for download at http://db.informatik.uni-mannheim.de/xdg.html



(a) Example document

Tag2Lid

Tag	Lid
A	1.1
A	1.1.1.1
A	1.1.1.1.1
...	...
A	1.1.5.3
A	1.1.5.3.1
B	1.1.1
B	1.1.3
B	1.1.5
DOC	1

Lid2Nid

Lid	Nid
1	1
1.1	2
1.1.1	3
1.1.1.1	4
...	
1.1.5	12
1.1.5.1	13
1.1.5.3	14
1.1.5.3.1	15

(b) Indices

Figure 5.2.: Indexing XML documents

indices are the better approach to evaluate path expressions. But before we discuss the plan alternatives and their performance, we introduce the structure of the indices, we employ in the plans we present in this section.

5.2.1. Indexing XML

Similar to the proposal of Chien at. al [CVZ+02], we index XML documents in B-link indices. As described in Section 2.3, we can reference physical nodes stored in our database with logical node ids (LIDs). We use ORDPATH IDs to identify logical nodes [OOP+04]. There exist other proposals for indexing XML documents, e.g. [GW97, LM01]. But we believe that our approach provides a solid performance for a wide range of queries and at the same time supports efficient concurrent updates.

The idea of our indexing scheme, depicted in Figure 5.2, is to create two indices: one called *Tag2Lid* and the other *Lid2Nid*.

Tag2Lid maps tag names to LIDs. The key value is the tag name, and the indexed value is the LID. For the same tag name, LIDs are returned in document order, i.e. in ascending order.

Lid2Nid maps LIDs to their physical storage address. This index is optional when the storage manager directly provides access to XML fragments based on their LID. However, for generality we will explicitly use this index to locate result nodes of an XPath expression.

We create the two indices shown in Figure 5.2(b) for XML document in Figure 5.2(a). The subscript of every node in the document is annotated with its LID. In Natix, we cluster

subtrees of the XML document on the same page to reduce I/O during document traversals and lookup of nodes, see [KM00, KM06] for details.

5.2.2. Query Execution Plans

The translation of path expressions into our algebra, described in Chapter 3, does not associate operator implementations with the axis steps. Moreover, the translation function schedules axis steps in their canonical order. It is the task of the cost-based optimizer to choose the most efficient order and implementation for axis steps. We now discuss the alternative query execution plans (QEPs) the optimizer may consider:

1. Navigate through the XML document (e.g. in a DOM-like fashion).
2. Use indices to access the candidate nodes of each navigation step and relate them by join operations to evaluate the query. If there are multiple navigation steps, we have two more choices:
   a) Access indices in the order specified in the query.
   b) Reorder the index accesses and sort the result nodes at the end.

In our opinion, these types of QEPs comprise a wide variety of XPath evaluation techniques that have not been compared yet. For each alternative mentioned above, we present a QEP for the query `/DOC/TAG1/TAG2`. Even this simple query allows us to point out the advantages of each alternative. The reason is that each QEP exploits structural relationships, selectivities of axis steps, or physical storage characteristics to different degrees. As we will see in our experiments, there is no single plan that is consistently faster than the other alternatives. Thus, the best plan must be chosen based on its estimated execution costs.

Plan Using Navigation

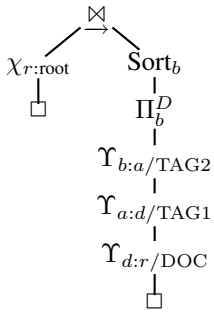


Figure 5.3.: Plan using navigation

The most direct translation of the XPath expression results in a navigational plan [BKHM05]. The result of the *stacked translation* of the query into our algebra is depicted in Figure 5.3. The topmost operator of the QEP is a D-Join which initializes the context for the XPath query evaluation to the root node. The right argument is evaluated with the bindings taken from this context. The stacked translation results in a sequence of Unnest Map operators, each of which evaluates one axis step. In general, to compute the resulting node set, duplicates have to be removed, and the result nodes have to be sorted by document order. In our example query, we can avoid duplicate elimination or sorting [HKM02, HM03].

When the QEP is evaluated, each Unnest Map operator traverses some part of the document, starting at the current context node. E.g. during a child step, all children of the current context node will be visited. When a node satisfies all node tests, it is passed to the next operator, where it may serve as another context node.

This evaluation strategy has three basic consequences: (1) Non-matching nodes may implicitly prune parts of the document from the traversal. Thereby, accessing physical pages is avoided for potentially large parts of the document. (2) Axis steps may visit intermediate nodes that will never be part of a matching path expression. E.g. for descendant steps, we have to look at all descendant nodes of the context node. (3) In Natix, the document traversal may visit a physical page in random order and multiple times.

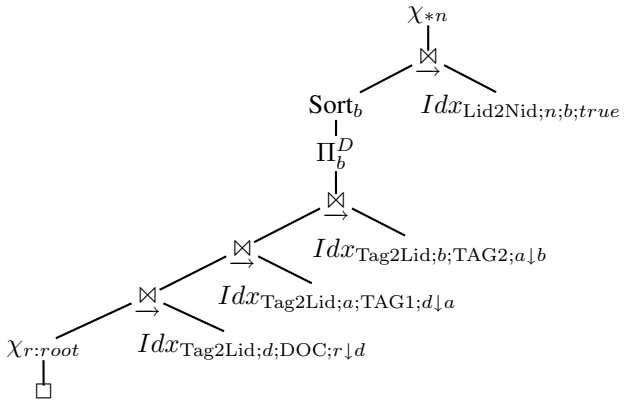


Figure 5.4.: Plan with index access in order

Plan with Index Access in Canonical Order

The motivation for using an index is to retrieve only nodes with tag names that satisfy the name tests of the axis steps in the path expression. The translation into a plan using an index is an application of the *canonical translation* presented in [BKHM05] or the XQuery translation of [PCS+05].

The result of this translation is shown in Figure 5.4. The data flow of the QEP goes from the bottom-left leaf node upwards to the root of the QEP. First, the root node is initialized as context node. This context can be used to restrict the range scan in the index "Tag2Lid". This index access is performed in the dependent part of each D-Join in the plan. We have to apply the residual predicate to each node retrieved from the index. Together with the range predicate, this test completes the structural test between context node and document node. Before all physical nodes are retrieved, we possibly have to perform a duplicate elimination and a sort [HM03]. Finally, we employ the index "Lid2Nid" to get the physical nodes of the query result and access the physical nodes on disk using a Map operator. Note that some queries do not require this final dereferencing step, e.g. quantified queries or queries with count aggregate. This can be used in favour of such queries.

The index-based technique has the following properties: (1) It only considers nodes which can match the node tests in the query. (2) The index is repeatedly accessed for each context node. This results in random I/O, as the same index page is accessed for different axis steps. (3) Context information can be used to prune the set of candidate nodes. This depends on the availability of e.g. level information for axis steps to sibling nodes. (4) Parts of structural queries can be answered solely based on LIDs. Hence, less information needs to be stored in the index. This potentially decreases the required I/O bandwidth. (5) Additional I/O is needed to retrieve the result nodes of the query.

Plan with Index Access Reordered

We now turn our attention to index-based QEPs in which we reorder axis steps. We treat the reordering of axis steps separately because there are two main issues that limit the value of join reordering for XPath expressions: (1) Join ordering in general is known to be NP hard. When we allow to sort by document order at the end, the search space contains $O((2n)!/n!)$ bushy join trees and $O(n!)$ left deep join trees [OL90, PGLK97] containing n joins. Here, we consider one scan for each axis step and include cross products. (2) The quality of the generated plans heavily depends on the precision of cardinality estimates [IC91]. However, good methods for cardinality estimation are known only for restricted classes of XPath [AAN01, LWP+02, PGI04, ZOAI06].

We can order the index accesses as shown in Figure 5.5. Reordering axis steps implies

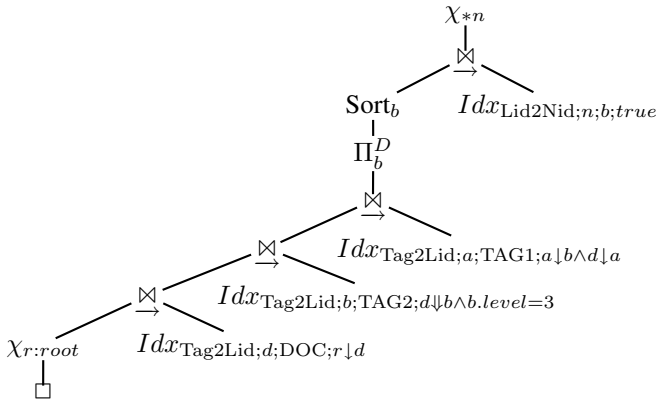


Figure 5.5.: Plan with index access reordered

three differences to the previous plan: (1) We reordered the axis steps. (2) The residual predicates had to be adjusted. (3) To establish the document order, we need a final sort.

The potential value of reordering axis steps stems from the possibility of evaluating axis steps with low result cardinality first to minimize the number of lookup operations in the index. The additional freedom of reordering axis steps has to be payed with an additional sort operation (which is always needed now) and less restrictive structural predicates. Hence, it is not clear which strategy is better in which case. In general, this decision should be based on costs.

Plan Using Index and Structural Join

The plans discussed in the previous sections access the index for each context node. Thanks to the Structural Join ($\bowtie_p^{ST-J}$), we can evaluate an axis step with a single scan of each input, and we still have the full freedom to choose the most efficient plan among all bushy join constructed with Structural Joins [SAKJ+02, WPJ03]. Notice that this is not generally true for staircase joins [GvKT03] One possible QEP using Structural Joins is depicted in Figure 5.6. In this plan, a Structural Join is performed between the nodes with tag name `TAG1` and `TAG2`. Since both input sequences are sorted in document order, the Structural Join can compute its result with one scan through both sequences and some additional buffering. The resulting node sequence is sorted by document order and does not produce duplicates.

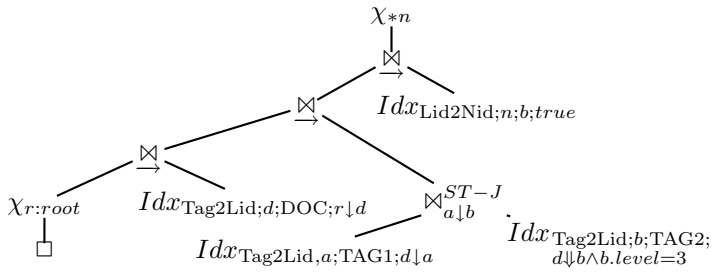


Figure 5.6.: Plan using index and Structural Join

5.2.3. Experiments

Now, we compare the performance characteristics of the QEPs developed in Section 5.2.2 for three XPath queries on the synthetic data set presented in Section 5.1.

We have executed all queries on Natix. Each query was executed three times with cold buffer (with 8MB buffer size). We report the average of all three evaluations. Our execution environment was a PC with two 3GHz Intel Xeon CPUs, 1 GB of RAM, 34GB hard disc (FUJITSU MAS3367NC) running SUSE Linux with kernel 2.6.11-smp.

XPath Queries

We have compared the performance characteristics of the QEPs discussed in Section 5.2.2 for the following three XPath query patterns:

Q1: `/descendant::TAG`. This query reveals the impact of the access patterns of the QEPs because when evaluating this query, structural information is unimportant. The navigational plan visits the whole document to access all potential result nodes. In contrast, the index-based plans only visit only the nodes with tag name `TAG`.

The main difference is that the navigational plan performs random I/O in the worst case, whereas the index-based QEP can directly retrieve the requested nodes by a range scan on the "Tag2Lid" index.

Q2: `/DOC/TAG1/TAG2`. With this query, we investigate (1) how well each plan alternative exploits structural properties demanded by the query, and (2) how reordering navigation steps effects query performance.

Q3: `/DOC/descendant::TAG1/descendant::TAG2`. In addition to query Q2, the cost of evaluating each step in this query is potentially much higher because level information is less useful here. As both descendant axis steps potentially visit large parts of the document, we expect optimizations that can reduce the I/O to be very important.

We restrict ourselves to the child axis and the descendent axis because only for these two axes precise selectivity estimation techniques are known, e.g. [AAN01, LWP+02, ZOAI06, PG06]. To make our experimental results comprehensible, we ignore the other XPath axes because we cannot easily compute the selectivity of an axis step with respect to some arbitrary context node.

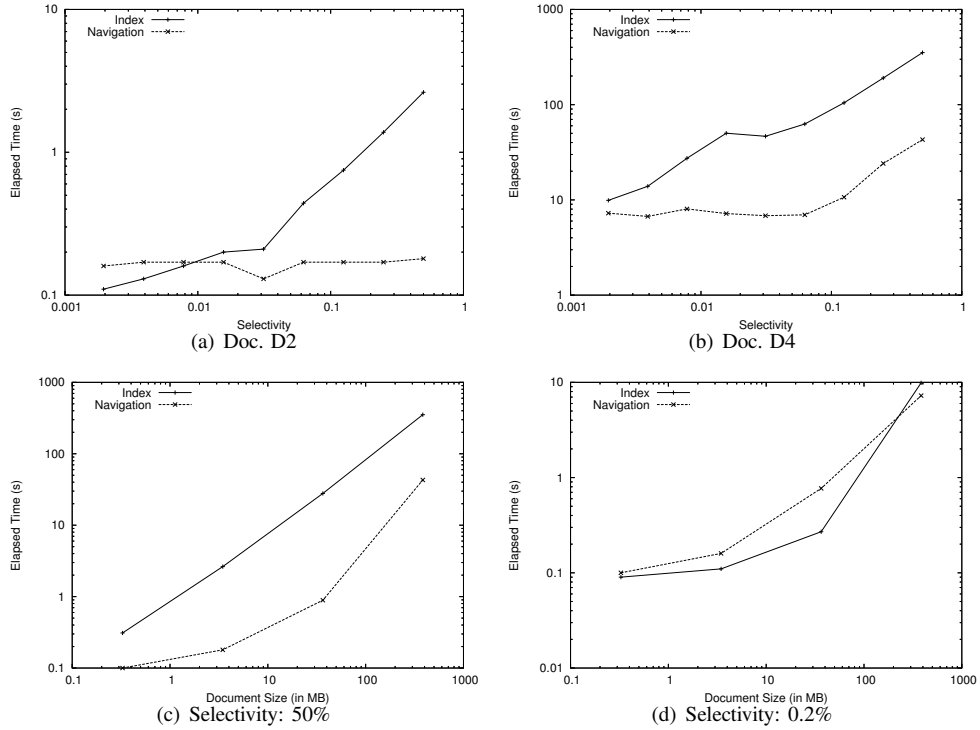
Experimental Results

Query Q1. Figure 5.7 shows the results for query Q1. In Figures 5.7(a) and 5.7(b), we compare the performance of the navigational plan and the index-based plan for document D2 (3.46MB) and D4 (384MB).

For small selectivities on the smaller document, the index-based plan performs better than the navigational plan. As we make the node test less selective, the index-based approach needs more time to evaluate the query, while the execution time of the navigational plan remains nearly constant. The break-even is reached at a selectivity of about 1%. For larger selectivities, the navigational plan outperforms the index-based plan. All these results agree with our experience in the relational world.

In Figures 5.7(c) and 5.7(d), we plot the query execution times for specific selectivities of 50% and 0.2% over documents D1, D2, D3, and D4.

Again, the results confirm that index-based evaluation is superior only for selective queries. However, since two indices and the document are accessed, more buffer pages have to replaced due to lack of space, and additional I/O is needed. As a result, the performance of the index-based plan suffers on the largest document instance.



(a) Doc. D2 (b) Doc. D4

(c) Selectivity: 50% (d) Selectivity: 0.2%

Figure 5.7.: Query Q1 (/descendant::TAG)

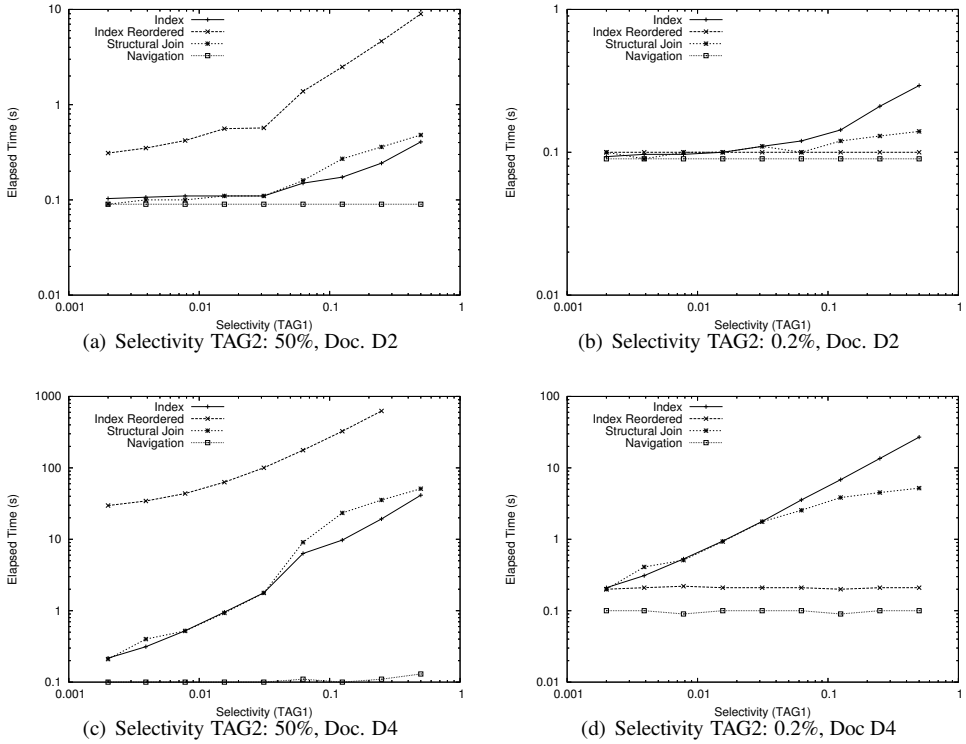
Query Q2. Figure 5.8 contains the results of our experiments for query Q2. We restrict ourselves to two document sizes (3.46MB and 384MB) because the results on the other documents do not provide any additional insight. In our exposition, we keep the selectivity of the second axis step (`TAG2`) constant at 50% (see Figures 5.8(a) and 5.8(c)) and at 0.2% (see Figures 5.8(b) and 5.8(d)). We only modified the selectivity of the first axis step (`TAG1`).

The navigational QEP has an almost constant execution time independent of the selectivity. This is a direct consequence of its evaluation strategy: This QEP has to inspect the same set of nodes to compute its result, no matter how selective each step is. The navigational QEP dominates all other QEPs because this plan only visits the three upper levels of the document.

The value of reordering axis steps becomes apparent when we compare the execution times of the naive and the reordered version of the index-based plans. In the experiments depicted in Figures 5.8(a) and 5.8(c), the reordered version is up to ten times slower than the naive plan because the reordered QEP performs the more expensive scan first (selectivity 50%). In Figures 5.8(b) and 5.8(d) we observe exactly the reverse behavior because the second axis step is very selective (selectivity = 0.2%). However, the differences are smaller because the naive plan can use more information to restrict the index scan. In all experiments the Structural Join behaves similar to the naive index-based evaluation technique. The index based technique is competitive because none of the navigation steps produces duplicates. Hence, no redundant index lookup is performed.

The advantage of the navigational plan is partially a consequence of the document structure. For shallow documents where we increase the number of children per node, we expect similar behavior as for query Q3.

Query Q3. For our final experiment, we replace the last two child steps of Q2 by descendant steps: `/DOC/descendant::TAG1/descendant::TAG2`. The execution times



(a) Selectivity TAG2: 50%, Doc. D2

(b) Selectivity TAG2: 0.2%, Doc. D2

(c) Selectivity TAG2: 50%, Doc. D4

(d) Selectivity TAG2: 0.2%, Doc D4

Figure 5.8.: Query Q2 (/DOC/TAG1/TAG2)

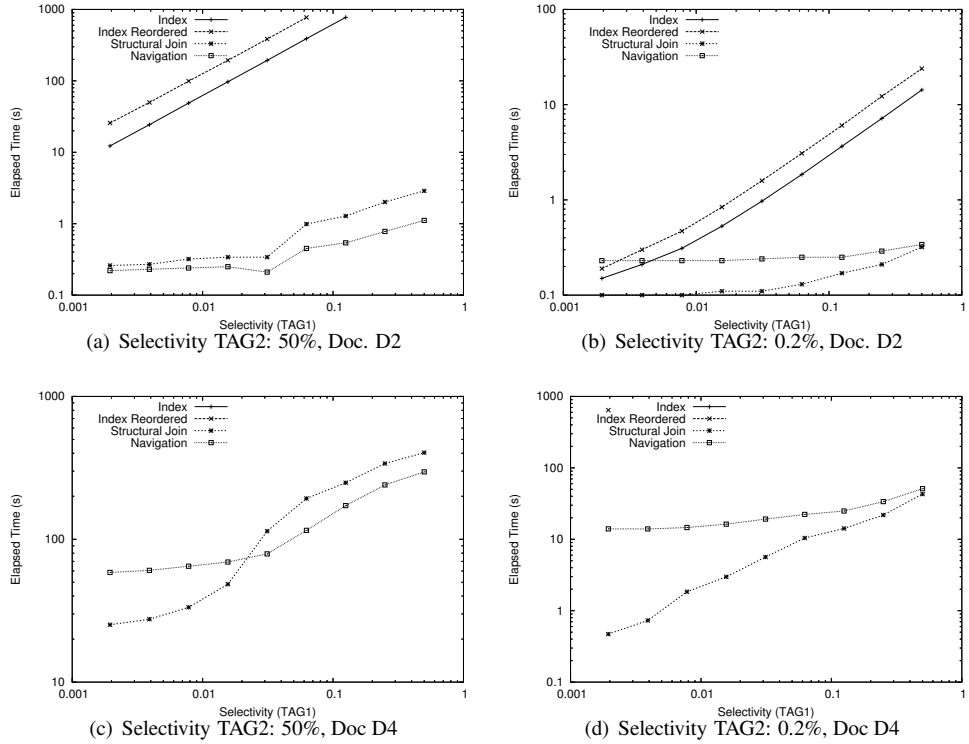
for these three alternatives are shown in Figure 5.9. In the results we present here, we use the same parameters as we did for query Q2.

First, we discuss the navigational approach: We observe that in all figures the evaluation times of the navigational QEP increase with an increasing selectivity of the name tests. The reason for this is that the XPath expression generates more context nodes for which the whole subtree is traversed. For larger selectivities, fewer subtrees are pruned during the traversal.

In contrast, the index-based QEPs retrieve exactly the candidate nodes with the correct tag name. However, only in Figure 5.9(b) both index-based strategies are faster than navigation for very small selectivities on a small document. The reason for this is that our naive index-based execution strategy is not aware of structural relationships of context nodes. As a result, the same nodes are returned repeatedly by index lookups. We conclude that the simple index-based QEPs do not result in an acceptable query performance for this query pattern.

The plan based on Structural Joins avoids these superfluous index lookups. For large selectivities (of 50%) of the second step, the Structural Join is faster when the selectivity of the first step is smaller than about 2% and the document is large (Figure 5.9(c)). However, when we keep the selectivity of the second step at 0.2%, the Structural Join clearly outperforms the navigational query independent of the selectivity of the first step and the document size. This advantage is larger when the selectivity of the axis steps is small.

Summarizing, our experiments show that neither navigation nor index-based plans are always superior. Moreover, there are cases where reordering axis steps is better than evaluating them in their canonical order, even when this requires an additional sort operation. Hence, all these alternatives are important and must be considered during cost-based plan generation.



(a) Selectivity TAG2: 50%, Doc. D2

(b) Selectivity TAG2: 0.2%, Doc. D2

(c) Selectivity TAG2: 50%, Doc D4

(d) Selectivity TAG2: 0.2%, Doc D4

Figure 5.9.: Query Q3 (/DOC/descendant::TAG1/descendant::TAG2)

5.3. Sequence Order Considered Harmful

When we combine sequences of items in XQuery by a series of **for** clauses, we have to respect the order of the combined sequences, i.e. their sequence order. Thus, the observable effect of nested **for** clauses must be the same as the evaluation of nested loops in which we iterate through the involved sequences.

In Chapter 2, we proved that joins over sequences are associative but not commutative. In particular, exchanging the arguments of a join destroys the order. Consequently, the query optimizer is severely limited when enumerating possible join orders. In this section, we show how the search space of a plan generator can be extended significantly and how this leads to better plans. We will allow to destroy order temporarily and add sort operations to repair order afterwards. Since sorting is not for free, the decision to destroy order should be based on costs.

With the help of the following example query, we show how extending the search space influences the quality of the QEP. We have chosen a simple SPJ-query that is well suited to explain the tasks performed by the plan generator during query optimization.

```
for $x1 in doc("d1.xml")//K,
    $x2 in doc("d2.xml")//A,
    $x3 in doc("d3.xml")//A
where
    $x1/*/@a = $x2/*/@a
and $x2/*/@d = $x3/*/@d
and $x1/*/@c = $x3/*/@c
return
  <result>
    <x1>{ $x1/@id }</x1>
    <x2>{ $x2/@id }</x2>
```

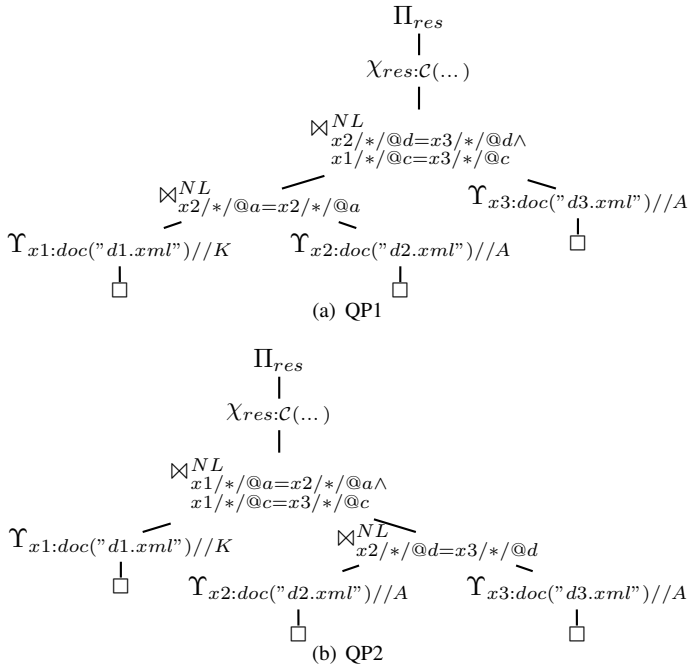


Figure 5.10.: Naive Plans

```
  <x3>{ $x3/@id }</x3>
</result>
```

Before delving into the details of optimizing this query, we explain some properties of the documents used in his query. They all have the structure of document D0 introduced in Section 5.1. The first path expression evaluates to a sequence of $\lfloor \frac{5000}{2^{11}} \rfloor = 2$ nodes, while the other two path expressions evaluate to approx. 2500 nodes. Thus, the variable $x1$ is bound to a sequence that is much smaller than the sequences bound to $x2$ and $x3$. Also note that the query graph contains a cycle $x1 \leftrightarrow x2 \leftrightarrow x3 \leftrightarrow x1$. The selectivity of the predicates also varies from the first to the last because in the first predicate only two different attribute values occur, as opposed to sixteen in the second predicate and eight in the last predicate.

5.3.1. Query Execution Plans

We now present several query execution plans (QEPs) to evaluate the example query. Starting with the canonical translation of the query that we have introduced in Chapter 3, we will increase the considered search space and discuss its impact. Since our focus here is on join ordering and to keep the QEPs readable, we omit many details on evaluating the path expressions involved. This is an orthogonal issue which we have examined in the previous section. Along the same lines, we abbreviate the element construction with $\chi_{res:\mathcal{C}(\ldots)}$.

Naive Translation

The semantics of XQuery demand that the result of a query is computed in document order. This means, when an element A is visited before element B in the traversal of an XML document, then A is located before B in the resulting sequence of elements. When sequences of items from multiple sources are combined using **for** clauses, the order of the **for** clauses in the query determines the order of the combined sequence of items.

The canonic translation of a query results in a sequence of (order-preserving) Unnestmap operators connecting the sequences in the **for** clauses. When an expression in the **for** clause can be evaluated independently of the previous ones, we can apply Eqvs 2.17 and 2.18 to transform them into Cartesian products. These Cartesian products can be turned into joins when we can have join dependencies between those sequences. Thus, a naive query translator will turn the example query into a sequence of nested loop joins (see query plan QP1 in Figure 5.10(a)). We employ nested loop joins due to their order-preserving property. Typically, hash join algorithms do not have this property because they partition the input on secondary storage. Sort-merge joins require their input to be sorted on the join attribute, which does not necessarily match sequence order.

Evaluating the join predicates is done via a nested query. For example, for the first join predicate all the attribute values of `$x1/*/@a` are generated and compared to those generated for `$x2/*/@a`. In case we find a value that both sequences have in common, the predicate is true. Another variant to evaluate the join predicates would be to attach the sequence of attribute values to each tuple of x1, x2, and x3 and employ an adapted version of a set-valued join algorithm [HM97, MGM03, RPNR00].

Ordering Order-preserving Joins

Based on the algebraic properties of the operators in a query, the plan generator may choose between different plans. For example, order-preserving joins are associative but not commutative. Compared to the general join-ordering problem, this results in a much smaller search space for join ordering. Let us denote by $C(n)$ the Catalan numbers. Then, for ordering n order-preserving joins, there are "only" $C(n) = 1/(n+1)\binom{2n}{n}$ different execution plans. Moreover, the join ordering problem can be solved in polynomial time ($O(n^3)$) independently of the query graph and the cost function [Moe03].

The choice of the best operator order depends primarily on the input cardinality and the selectivity of the joins. For our example query the plan generator may choose between two different queries, QP1 and QP2 (see Figure 5.10). When running the queries on Natix, QP1 takes 587.75 seconds, while QP2 takes 395.18 seconds.

As we can see, there is a better alternative to the naive translation of our query. This resembles results obtained by Wu et al. in [WPJ03] in the context of structural joins for evaluation of XPath. However, this is just a first step in extending the options of the plan generator. In the following sections, we present some approaches to increase the search space even further.

Disregarding Order Preservation

The cardinality of the input and the selectivity of the join predicate are also important parameters for generating the cost-optimal order of joins that do not preserve order. A rule of thumb for plan generators is, e.g., joining the input with the smallest cardinality and the most selective join predicate first. However, the naive order-preserving evaluation limits our choices significantly. In order to lift these restrictions, we now disregard sequence order during query processing. As a consequence, for n join operators we now have $(n+1)!C(n)$ possible orderings (due to the commutativity of a join operator that does not preserve order) [OL90, PGLK97]. Thus, we get twelve different plans for our example query, increasing our search space considerably. Another consequence is that we have to sort the final result to make sure that sequence order is obeyed, since it may have been corrupted by using non-order-preserving join operators or by reordering the joins exploiting the commutativity.

Let us examine some implementation details of the plan depicted in Figure 5.11. First of all, we use a tid operator to add an attribute containing the position to each tuple in a sequence. This way, we are able to reconstruct sequence order later on. We also unnest the join attributes. For example, for the first join predicate this means that we generate a tuple

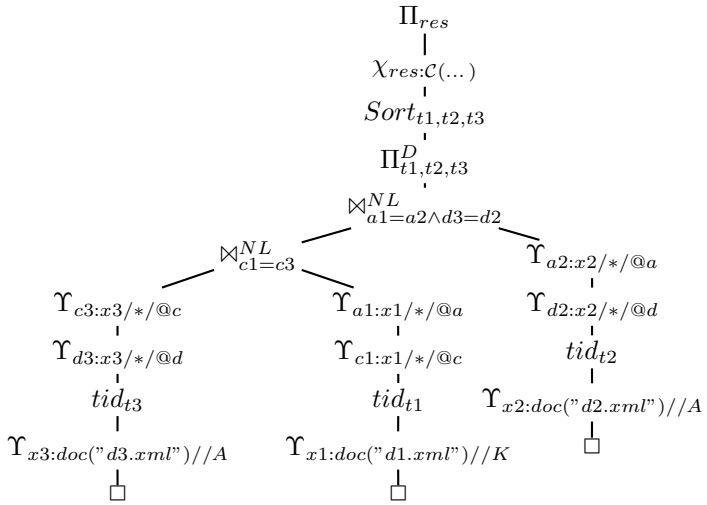


Figure 5.11.: Optimized Join Order (QP3)

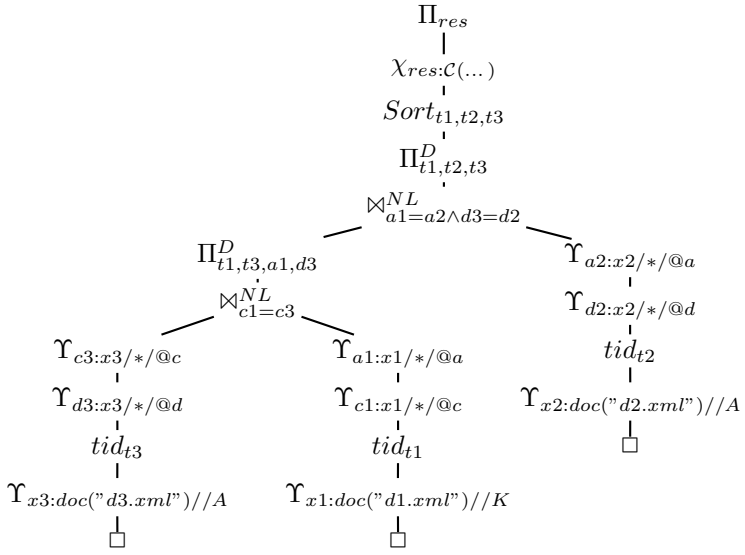


Figure 5.12.: Optimized Join Order with pushed duplicate elimination (QP3_de)

for each attribute value in `doc("d1.xml")//K/*/@a` and `doc("d2.xml")//A/*/@a` and then join the two sequences on these attribute values. As this results in duplicates, we also have to insert a duplicate elimination step after joining all sequences.

The best QEP we found among all twelve plans, called QP3, is depicted in Figure 5.11. In this case, the sequences for x1 and x3 are joined first. In our case, this is better than joining the smallest sequence with the least selective predicate or joining the largest sequences with the most selective predicate first. Comparing the execution time of this plan (125.48 seconds) to QP2 from the previous section, we see that we have improved the performance approximately by a factor of three.

Pushing Duplicate Elimination

As noted in the previous section, the joins might contain duplicates in their results. We now extend the search space of the plan generator to consider introducing an additional duplicate elimination after each join. In a query with n joins, we may introduce at most $n - 1$ additional duplicate elimination operators. The final duplicate elimination is mandatory in our case. Note that introducing the duplicate elimination does not harm the sequence order.

Duplicate elimination does not come for free. It is not trivial to decide if introducing duplicate elimination is beneficial, and this issue has been addressed in the context of *early aggregation* in the literature [Lar02]. Deciding on the effectiveness of doing so is the task of the cost-based query optimizer [HNM03].

In our example, the search space is extended only by one additional alternative for QP3. The resulting plan QP3_de, shown in Figure 5.12, contains an additional duplicate elimination, $\Pi^{D}_{t1,t3,a1,d3}$, after the join of $x1$ and $x3$. Since this duplicate elimination reduces the input cardinality of the last join, the execution time of this plan improves to 103.21 seconds. As the example query shows, it is worth to extend the search space of the plan generator to consider introducing duplicate elimination. However, it is not always worth doing this extra work, but it gives more freedom to the plan generator.

Choice of Join Algorithm

When giving up the order-preserving property of the join operators, we can go one step further and employ different non-order-preserving implementations of join operators. Consequently, we extend the search space even more by allowing the plan generator to choose among several different join evaluation techniques [DKO+84, Gra94, GLS94, Gra93, HCLS97].

In our case, we implemented two other join algorithms: a hash-based and a sort-based one. The SIMPLEHASHJOIN is a block wise nested loop algorithm that pipes main-memory sized blocks of the outer producer into a hash table and probes each block with all the tuples of the inner producer. The sort-based MERGEJOIN is a standard n:m sort-merge join.

For our measurements, we restrict ourselves to the most efficient plan from the previous section. We replace the nested loop joins either by hash joins or sort-merge joins. For the hash join we have a query evaluation time of 7.34 seconds and for the sort-merge join an evaluation time of 8.00 seconds.

5.3.2. Performance Summary

The comparison of the query execution plans in this section show that extending the search space of the query optimizer leads to substantially more efficient plans.

The execution times of the plans we have considered here are summarized in Figure 5.13. The canonical translation of the query resulted in a plan that needs almost ten minutes to join the three documents of about 0.4MB size. Exploiting associativity, we could improve execution time to about 6 minutes which we consider as not acceptable. This improvement still represents the state of the art of join ordering for XQuery. By disregarding order, we improved query execution times to two minutes. The main advantage of disregarding order is that we are now free to choose an efficient implementation for performing the joins. In our case, both hash join and sort-merge join evaluate this query in about eight seconds. Comparing this to the original optimal order-preserving plan, we improved the execution time by almost 50 times even on this small example query with just two join operators!

5.4. Towards Cost-based Optimization of XQuery

XQuery still has lots of potential for query optimization, in particular when several query execution plans are enumerated and the cheapest is chosen among all alternatives. In the

Plan	Join Algorithm	Preserve Order?	Dupl. Elim.	Time (s)
Q1:	NLJOIN	Yes	No	587.75
Q2:	NLJOIN	Yes	No	395.18
Q3:	NLJOIN	No	Final	125.48
Q3_de:	NLJOIN	No	Pushed	103.21
Q3_de:	MERGEJOIN	No	Pushed	8.00
Q3_de:	SIMPLEHASHJOIN	No	Pushed	7.34

Figure 5.13.: Comparison of query execution times

previous two sections we have demonstrated that reordering operators can lead to substantially more efficient query execution plans. When we detect that reordering some operators does not preserve order, but is valid for bags, we need to insert a sort operation to repair order later. Since sorting is a costly operation, this decision should be based on costs.

As a consequence of these observations, an efficient management of orders (and possibly also of duplicates) becomes a key requirement for cost-based XQuery optimizers. The main idea of our approach is to use interesting orders [SAC+79] to model both document and sequence order explicitly. As outlined in Section 5.2, logical node ids allow us to check if two XML nodes are in document order. On the other hand, we can employ the tid operator to make sequence order explicit (see Section 5.3). Then, we can use the tids both to repair sequence order and to remove duplicates. In our query optimizer, we rely on an efficient management of interesting orders [NM04].

In [Hac06], we have generalized the concept of interesting orders to interesting properties in general. In the remainder of this chapter, we describe how we integrate this general property framework into our cost-based query optimizer. As a result, we can efficiently generate query execution plans that exploit all the ideas discussed in Sections 5.2 and 5.3.

5.4.1. A Classification of Properties

The input to the cost-based query optimizer consists of a logical expression, i.e. the query representation used before this step. This means, until this phase the query representation does not contain any implementation details. Besides the operators contained in this query representation, properties of this query representation are important information when generating and comparing plan alternatives. Consider the following query:

```
for  $x1 in  //K/@a,
     $x2 in  //A/@b,
     $x3 in  //A/@a
where
     $x1 eq $x2
and $x2 eq $x3
return
  <r> { $x1, $x2,  $x3 } </r>
```

The translation of this query into an algebraic expression results in:

$$
\begin{aligned}
& \Pi_r(\chi_{r:\mathcal{C}(\ldots)}(\sigma_{x1=x2\wedge x2=x3}(\Upsilon_{x3://A/@a}(\Upsilon_{x2://A/@b}(\Upsilon_{x1://K/@a}(\Box)))))) \\
\overset{(2.17)}{=}\ & \Pi_r(\chi_{r:\mathcal{C}(\ldots)}(\sigma_{x1=x2\wedge x2=x3}(e_3 \underset{\rightarrow}{\bowtie} (e_2 \underset{\rightarrow}{\bowtie} e_1)))) \\
\overset{(2.18)}{=}\ & \Pi_r(\chi_{r:\mathcal{C}(\ldots)}(e_3 \bowtie_{x2=x3} (e_2 \bowtie_{x1=x2} e_1)))
\end{aligned}
$$

where

- Joined data sources
- Applied operators
- Result cardinality
- Tuple size
- Key
- Functional dependency

(a) Logical Properties

- Cost
- Order
- Grouping
- Duplicate Free
- Site in a distributed DBMS
- Presence in main memory

(b) Physical Properties

Figure 5.14.: Examples of logical and physical properties

$$
\begin{aligned}
e_1 &:= \Upsilon_{x1://K/@a}(\square) \\
e_2 &:= \Upsilon_{x2://A/@b}(\square) \\
e_3 &:= \Upsilon_{x3://A/@a}(\square)
\end{aligned}
$$

Remember that the task of the cost-based query optimizer is to find the cheapest order and implementation of the operators in the query. Logical and physical properties that hold for an argument plan are important information when constructing and comparing plan alternatives.

Logical Properties need to hold for every plan considered by the plan generator, independent of the operator implementations involved. In our example, the join $e_1 \bowtie_{x1=x2} e_2$ requires that the subexpressions e_1 and e_2 are completely contained in the argument plans of this join. Otherwise, attributes $x1$ and $x2$ in the join predicate are not available.

When the plan generator considers to use a sort-merge join to implement the above join, it must additionally hold that expressions e_1 and e_2 are sorted on the values in $x1$ and $x2$, respectively. Since sequence order and the order of the values do not coincide in general, an additional sort operation on e_1 and e_2 is needed to establish the order required by the sort-merge join. The sort operator *enforces* the order property required by the sort-merge join. *Physical Properties* are properties that are required or produced by specific algorithms used in a plan.

Hence, physical properties of a plan depend on the operator implementations used, whereas logical properties do not. In this work, we follow the classification of plan properties proposed by McKenna [McK93] and distinguish logical and physical properties. Examples of them are summarized in Figure 5.14. We also use the term enforcer introduced there to establish required physical properties. A similar classification is proposed by Lohman [Loh88], where enforcers are called glue operators.

With the advent of XPath and XQuery, several new properties were invented. Helmer et.al [HKM02] use them to avoid creating duplicates. Hidders and Michiels [HM03] use an automaton to decide if after application of an axis step, the result is still free of duplicates and in document order.

If not, they insert enforcers to remove duplicates or establish document order. Interestingly, their automaton derives these physical properties based on a set of logical properties that hold for the path expressions. Grust et. al [GRT07] are able to derive when order is not relevant. As a result, order constraints on the query result can be relaxed.

It is likely that new properties will be discovered. For example, new properties may be able to detect conditions when physical properties need not be enforced. Or new algorithms to implement database operations may demand properties yet uncovered to hold. This is our motivation to develop an extensible framework to support an arbitrary number of logical and physical properties efficiently.

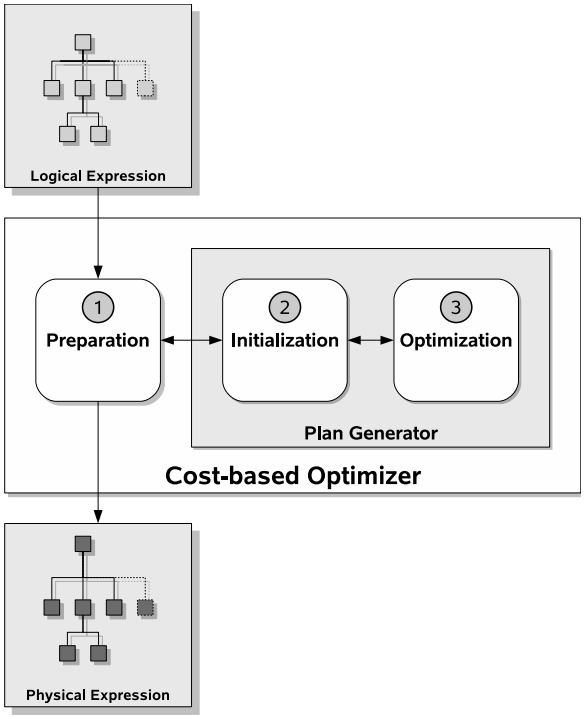


Figure 5.15.: Process of cost-based optimization [Hac06]

5.4.2. Generic Property Support for Plan Generators

Now, we give details how the properties identified in this chapter are communicated to the plan generator. As shown in Figure 5.15, the plan generator is the main component of the cost-based query optimizer. Since the plan generator considers many plan alternatives in the optimization phase, a space-efficient representation of plan alternatives with time-efficient access methods is of utmost importance. In contrast, the query representation is less sensitive to these issues, but usability of properties is the main concern. Consequently, we distinguish the three phases in Figure 5.15. Among other things, the first two phases gather properties and convert them into more efficient representations used in the last phase, optimization. In our presentation, we concentrate on document order, sequence order, and duplicates because they are the most relevant properties for XQuery processing. A more general treatment of properties and their implementation can be found in [Hac06].

Preparation Phase

We concluded in Sections 5.2 and 5.3 that an explicit representation of both document order and sequence order is needed to restore order after it was destroyed temporarily. Until cost-based optimization, we only apply rewrites that preserve order. Hence, we do not need to care about this issue there. Now, during cost-based optimization, we allow reordering axis steps or joins. Thus, we need to be able to detect that order was destroyed relying on an explicit representation of order information.

The first key observation we use to preserve document order is the following: We have to assure that a path expression returns its final result in document order and duplicate free. Therefore, we can exploit the fact that all nodes in our system are associated with a logical node id (LID) that can be used to compare if two nodes obey to document order. Consequently, we explicitly request the node sequence resulting from a path expression to

be in document order. If the cheapest plan constructed in the optimization phase preserves order, this plan will not contain a sort operator to repair order. On the other hand, explicit sort operators are inserted automatically to satisfy the sorting request. Similarly, LIDs allow us to detect duplicates.

The second key observation we use to preserve sequence order is that we can employ the tid operator to achieve the same effect for sequence order as LIDs do for document order. Hence, our approach is to add new attributes representing tids whose value depends on the computation of the whole path expression bound in the **for** clause. We annotate every tid added this way as sorted in ascending order and duplicate free. Finally, we request the final result of the plan to be sorted by the tids in ascending order and to be duplicate free. If we have multiple **for** clauses in the query, we add a tid for every clause. The resulting sort request is the compound order with major order prescribed by the tid of the first **for** clause and the least significant order given by the last **for** clause.

Alternatively, we could also use the sort key of every path expression to create a larger compound sort key for sequence order. Of course, this only works for sequences of nodes. But it avoids adding new attributes and trades reduced memory consumption for more complex sort keys.

In the preparation phase, we also compute information used for scheduling operators. Therefore, we annotate every operator with the operators that are required in an argument plan and the operators that are forbidden in an argument plan. This allows us to control scheduling of algebraic operators beyond their immediate argument relationships. Currently, we fix grouping, semijoin, antijoin, and outerjoin at the position that resulted from unnesting. But the performance gains possible from reordering these operators makes reordering them a desirable future extension of our optimizer [GLR97, RLL+01, YL94].

Summarizing, we use the preparation phase to extract all operators from the query representation. We detect path expressions and demand their results to be in document order and duplicate free. Similarly, we materialize sequence order in auxiliary attributes and use these attributes to assure proper ordering of the final result.

The actual implementation of the preparation phase maps objects in the query representation to numbers. This prepares the query representation for an even more compact representation which is constructed in the initialization phase, the next phase. This mapping of operators to numbers is used to extract the final query execution plan from the plan generator.

Initialization Phase

The preparation phase has gathered all property information and all operators from the query representation. The initialization phase examines this information and (1) derives further information from the input to avoid repeated computations in the optimization phase, (2) creates an extremely compact representation of any kind of information used in the optimization phase, e.g. operator identification or available properties.

A novelty we introduce in this phase is that collecting all physical properties required by an implementation rule is carried out by the same rule. This design is motivated by the observation that the (optimized) information is also used by these rules in the optimization phase. We expect that this design allows extensions to new properties or operators to be integrated more effectively.

Property computation focuses mainly on physical properties because all logical properties are already contained in the query representation. On the other hand, physical properties thus far only include the orders demanded in the **order by** clause, duplicate elimination (e.g. `fn:distinct-values`) of the XQuery statement, and the sorting and duplicate elimination requests generated in the preparation phase.

But many more physical properties may be relevant during cost-based query optimization. Resuming with the example query introduced in Section 5.4.1, the rule for the MERGEJOIN adds orders on its join attributes as relevant physical properties. After query

unnesting, the query may contain grouping operators. Thus, the rule for sort-based grouping adds an order on the grouping attributes as interesting order. Duplicate elimination is implemented with grouping. Hence, sets of attributes that are duplicate-free are also accumulated in this phase. For example, more efficient implementations of the MERGE-JOIN exist if at least one input does not contain duplicates on the join attributes. Moreover, grouping comes for free when we can assure that the grouping attributes do not contain duplicates.

All this information is gathered by iterating over all operators contained in the query. For every operator and every rule that implements this operator, the function `Rule::init` is called, which adds all physical properties.

In the next step, the collected physical properties are mapped to space-efficient representations that also support all required operations on them efficiently. In [NM04] the mapping we use for order and grouping is described. This preprocessing step has the following positive impact on the optimization phase: (1) It avoids repeated computations, e.g. if an order is subsumed by another. (2) It makes operations faster, e.g. an equality test of orders maps to a pointer comparison. (3) It saves space, by using bitmaps or pointers to singleton instances of a property value wherever possible.

Optimization phase

The optimization phase enumerates all possible query execution plans; implementation details can be found in [Neu01]. Currently, we use the enumeration technique of [VM96] to generate QEPs in a bottom-up fashion effectively performing dynamic programming.

When a new (partial) plan is constructed, the implementation rule first checks if all required logical and physical properties are fulfilled, e.g. all required operators are contained in the argument(s), and no forbidden properties hold for the argument plans, e.g. no forbidden operators are already scheduled. If all tests are passed, a new (partial) plan is constructed and the resulting logical and physical properties are derived. Missing physical properties are established by scheduling enforcers. Below, we discuss how we avoid some pitfalls when we want to enforce multiple properties. The new plan is added to its plan class and cost-based and property-based pruning is performed.

To support XQuery, we treat axis steps explicitly as new operators. This is required to accurately derive all physical properties for different implementations and different axes. For example, certain combinations of axis steps potentially produce duplicates or potentially destroy document order [HM03]. But other implementations always produce their result in document order and without duplicates if their input has these properties. Notice, however, that document order and sequence order do not require specific treatment beyond the general framework.

Enforcing Multiple Properties

Enforcers are used to establish physical properties. So far, the integration of enforcers into the plan generator was done in an ad-hoc fashion because enforcers such as Sort or Ship (to another site in a distributed system) did not interfere.

Now that we consider more enforcers and enforcer implementations, we have to consider that enforcers interfere. On the one hand, this can be a problem, e.g. when a hash-based duplicate elimination destroys the order established by a sort operation. On the other hand, it can be an advantage, e.g. when a sort-based duplicate elimination can benefit from an existing order. The issue of interfering enforcers has not been investigated yet.

To formalize this problem, we have to model the dependencies between enforcer implementations. We use the following notation to describe the behavior of some operator o with respect to some property p:

o req p operator o requires property p

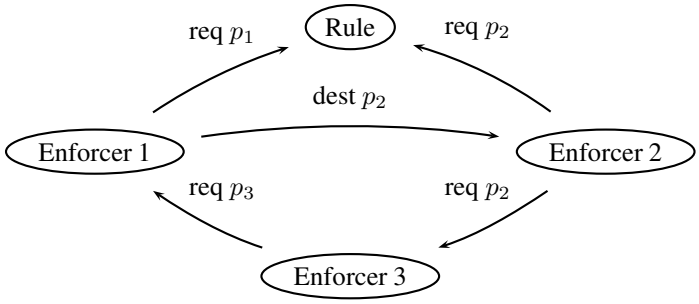


Figure 5.16.: Example for enforcers

o dest p operator o destroys property p

o prod p operator o produces property p

Now, for every implementation rule of some logical operator we construct a rooted directed graph $G(V, E)$ with the following properties:

1. $r \in V$ is the root of the graph and denotes the considered implementation rule.
2. $v \in V$ for every enforcer that produces any required property, including those required by r.
3. $\forall x, y \in V : (x, y) \in E \Leftrightarrow (x \text{ prod } p \wedge y \text{ req } p) \vee (x \text{ dest } p \wedge y \text{ prod } p \wedge r \text{ req } p)$.

The third rule states that an edge is added to G if one of the following two conditions holds. (1) If enforcer x produces property p and operator y requires p, then x has to be scheduled before y. (2) If enforcer x destroys property p, enforcer y produces p, and operator r requires p, then x has to be scheduled before y.

A valid order in which every enforcer is scheduled at most once can be obtained by a topological sort of the directed graph iff G is acyclic. Among all possible topological sorts, the plan generator should select the cheapest order. If G contains a cycle, we can remove edges from G to remove cycles, do a topological sort, and try to schedule an enforcer twice. We suggest to remove edges first that represent the destruction of some property. If this does not help, the combination of required physical properties might be pathological. Particularly, we believe that cycles containing only edges that are annotated with requirements cannot be satisfied at all.

Consider Figure 5.16 for an example. The rule requires properties p_1 and p_2. Enforcer 1 requires enforcer 3 to produce property p_3. At the same time, this enforcer 1 destroys property p_2, and enforcer 3 requires property p_2. Evidently, these three enforcers are part of a cycle. Therefore, we cannot simply use the result of a topological sort to order these operators. But if we ignore the edge labeled "dest p_2", a valid order of these rules is: Enforcer 2 → Enforcer 3 → Enforcer 1 → Enforcer 2 → Rule.

We now investigate the problem of finding the most efficient application order among all possibilities, assuming that the dependency graph G is acyclic. For simplicity, we assume that no properties are destroyed by any of the enforcer implementations and only one enforcer implementation is provided per property.

Let us denote with $n = |V|$ the number of vertices (i.e. the number of enforcers) in G. In the worst case, $O(n!)$ different topological application orders can be chosen. This is the case if no dependencies are defined among the corresponding enforcers. Thus, any permutation of the individual enforcer implementations can be used.

When some implementation rule requires physical properties, we have two basic choices to schedule the enforcers that establish these properties. First, the dynamic approach may

use a greedy algorithm (heuristic) or enumeration (optimal) to schedule enforcers on demand. The greedy heuristic performs a topological sort in $\Theta(|V| + |E|)$ time. When multiple orders exist, we schedule the cheapest enforcer first. The cost-optimal order of the enforcers can be computed be enumerating all $n!$ possible orders of enforcers. But using memoization, the runtime can be reduced to $O(2^{n-1})$, analogous to join ordering of left-deep join trees [Moe05, OL90, PGLK97]. If G is a dense graph resulting in few valid topological sorts, then the algorithm of [VR81] might be more efficient. It enumerates all valid topological sorts with respect to the partial orders given to the algorithm.

Second, when we can assure that no combination of required properties generates a cyclic dependency graph, it is also possible to hard-code a preferred order of enforcers. In this heuristic solution, the best order of enforcers is decided by the implementor using either of the algorithms above.

Rules of enforcers in our plan generator are no different from other implementation rules. In particular, different implementations for the same logical enforcer might exist. All enforcer rules derive from a common base class `Enforcer`. In some cases, the actual implementation of an enforcer rule simply refers to the implementation rule of a logical operator. For example, our duplicate elimination simply refers to the implementation rules for grouping. Nevertheless, we could also employ specialized algorithms for duplicate elimination [BD83, HNM02].

5.5. Plan Polishing

The query execution plan constructed during cost-based optimization can often be improved in a subsequent heuristic rewriting step called plan polishing. This phase is motivated by the following observations: (1) Some very specific implementations for query patterns may not be considered during cost-based optimization to avoid the involved complexity. (2) Adjacent operators can be merged.

In our current implementation, we focus on the latter issue. In particular, we implement equivalences that merge operators when the involved functions or predicates are not sensitive to the position in the input sequence or its size. Their correctness follows directly from the proofs of the equivalences presented in Chapter 2.

$$\sigma_p(\sigma_q(e)) = \sigma_{p\wedge q}(e) \tag{5.1}$$

$$\sigma_p(e_1 \times e_2) = e_1 \bowtie_p^{NL} e_2 \tag{5.2}$$

$$\sigma_p(e_1 \bowtie_q e_2) = e_1 \bowtie_{p\wedge q} e_2 \tag{5.3}$$

$$\chi_f(\chi_g(e)) = \chi_{f\circ g}(e) \tag{5.4}$$

In addition to the equivalences above, we also push selections into scans and map operations into grouping. The benefit of merging operators is a reduced effort in the subsequent code generation phase [May02] and, most importantly, fewer function calls during query execution.

We plan to implement rewrites that detect the query patterns for grouping proposed in [WM99]. Their application is always beneficial because they reduce the number of scans or the memory consumption of the query execution plan.

5.6. Related Work

In this chapter, we have argued for cost-based optimization to generate optimal plans for XQuery statements. To support our claim, we have considered path expressions and joins as the two most performance-critical core features of XQuery. Below, we relate our findings to other optimization techniques that were proposed for XPath, XQuery, and join ordering.

Furthermore, we have outlined the architecture of our cost-based optimizer. It employs a powerful property framework to implement the functionality needed to realize the full optimization potential of XQuery. We relate the architecture to the conventional design of relational optimizers and survey the role of properties in query optimizers.

XML Storage

When XML documents are stored in a relational database, the logical tree structure of the documents must be shredded (are stored as BLOB) to map it into the relational data model [Gru02, TDCZ02]. To be able to restore this logical tree structure, either a labeling scheme or some explicit representation of parent-child references is used. Traversing the XML document requires join operations [ZND+01]. Since this is a performance-critical operation, algorithms were developed that exploit specific knowledge of the tree structure of XML documents. Below, we enumerate several such algorithms.

Native XML storage managers as the one used in Natix [FHK+02, KM00, KM06] or by IBM [OCP+05] are able to directly access the logical tree structure stored on disk. Basically the same holds for main-memory representations [RSF06] and object stores [JAKC+02].

Indexing XML

Indices, in particular B-trees, are a core access path for relational databases. To support updates and secondary indices, relational databases rely on the concept of tuple identifiers that serve as a logical identifier for the tuple physically stored on disk. Thereby, references to the logical tuple are effectively separated from its physical storage location.

To support efficient processing of path expressions in object-oriented databases, path indices were developed, e.g. [KM90, CCM96, FS98]. Since path expressions are at the heart of XPath and XQuery, they were also employed there [GW97, MS99]. Other indices used to accelerate the evaluation of path expressions include [CSF+01, LM01].

These early proposals for indexing XML did not address the problem of updates. Numbering schemes all require the renumbering of all nodes after a certain amount of updates. Based on the concept of Dewey IDs, space-efficient labeling schemes for XML were developed [CKM02, OOP+04]. They have the following desirable properties: (1) They avoid relabelling the nodes in an XML document and are still space efficient when the XML document is updated. (2) They separate the physical storage address from the logical node. (3) It is possible to test the structural relationship of nodes in the same document based on the label. Hence, we use ORDPATH IDs [OOP+04] in our native XML store and our indices. ORDPATH IDs in XML database systems play the same role tuple identifiers play in relational databases.

A cost-based selection of the access method is another issue. In proposals of an index only the effectiveness of the proposed index is compared to other index strutures for a limited set of queries. The results presented in this work question the universal advantage of indices for processing path expressions.

Nevertheless, in DB2 [BEH+06] indices are selected based on costs to filter documents or regions of a document that contribute to the query result. But the final result of a path expression is computed by the XNav operator. Another cost-based approach to access path selection is presented in [HDN+03].

Summarizing, many evaluation strategies for path expressions are tightly coupled with storage structures and specialized indices. We refer to [Wei06] for a comprehensive overview of indexing and labeling schemes for XML. But no satisfying strategy to select the most efficient access structure has been proposed yet.

Optimization of XPath

XPath query processing is approached by translating the statement into an algebra [PCS+05, BKHM05, RSF06] or tree patterns [JLST02]. Besides the statistical learning techniques presented in [ZHJ+05], we are not aware of any cost-based optimization approach to XPath. Instead, heuristic rewrites are applied to optimize the path expression.

For example, tree pattern minimization [Woo01, ZO02, AYCLS01] reduces the number of axis steps to perform. These optimizations are beneficial under the assumption that reducing the number of axis steps reduces the effort of processing the path expression. Notice, however, that the opposite idea, i.e. introducing new axis steps, may also yield better performance [PMC02].

Another interesting rewriting technique replaces occurrences of backward axes in path expressions by forward axes [OMFB02]. The rewrites enable streaming processing of XPath. It would be interesting to know if (at least some of) the rewrites improve the efficiency of query execution.

Query containment can be used to check the eligibility of materialized views for path expressions [BOB+04]. Using materialized views avoids the expensive evaluation of path expressions.

All optimizations we are aware of carefully avoid to destroy document order during query processing [WPJ03, PCS+05, BEH+06]. In particular, Hidders et al. [HM03, FHM+04] use automata to decide when duplicate elimination or sorting is actually needed. We are the first to explore the opposite approach [MHKM04]: We consider the cost of an additional sort operation to repair document order after we discard order temporarily. This allows us to reorder axis steps. Our experiments show that reordering axis steps can improve execution times substantially.

Evaluation Techniques for Path Expressions

Several algorithms to evaluate path expressions were proposed. The structural Join [SAKJ+02], the Staircase Join [GvKT03], the Holistic Twig Join [BKS02], or the XNav operator [JFB05] are efficient implementations of path expressions. The latter two operators work on a coarsely grained level because they evaluate complete tree patterns. They do not easily allow to switch between navigation and their specific evaluation techniques. Additionally, no algebraic equivalences are known for them to be used for optimizing XPath.

Structural join algorithms rely on an efficient test of the structural relationship between context nodes and candidate result nodes which is usually accomplished by a node labeling scheme. It is a natural extension to combine index access with processing structural joins, as it is done in [CVZ+02]. We have used these ideas in our query execution plans for path expressions.

The XPath processing algorithms mentioned above are most effective if tree patterns are large. This leads to the necessity to detect tree patterns in an XQuery statement [JHSV06]. Once the tree pattern is detected, a (cost-based) decision should choose the most efficient processing strategy [HDN+03, MMS06, MBB+06].

Given a sequence of context nodes, evaluating an axis step might produce duplicates. These duplicates are the source of exponential execution times observed for some XPath processors [GKP02, GKP03b]. Gottlob et al. [GKP02, GKP03b] solve this problem using memoization avoiding repeated computations. However, their algorithm requires $O(|D|^3 + |Q|^2)$ space, where $|D|$ is the size of the document. This rules out documents that are significantly larger than physical main memory.

In Natix, we avoid this space overhead using pipelining and early removal of duplicates [BKHM05]. As an alternative, Helmer at al. [HKM02] avoid generating duplicates by introducing variants of XPath axis steps. Hidders et al. [HM03] use automata to decide when duplicate elimination or sorting is actually needed. However, it is not clear if removing duplicates and sorting is always the most effective evaluation strategy. We propose to

base this decision on costs.

Optimization and Evaluation Techniques for XQuery

In Lore, one of the first systems to work with semistructured data, a cost-based optimizer is used to generate the query execution plan with lowest I/O costs [MW99]. Possible query processing strategies include bottom-up (using one of the indices), top-down (navigation), and hybrid combining those two strategies. These basic evaluation techniques were developed in the context of the query language Lorel, a predecessor of XQuery.

Halverson et al. [HDN+03] also do cost-based optimization minimizing I/O costs. Since their data is stored solely in B^+-tree indices, the optimization task does not differ from optimization in relational databases. It would be interesting to transfer these ideas to systems with native XML storage managers. However, estimating the I/O operations for native XML stores is still an open problem; see [ZHJ+05] for one proposal. Currently, we derive cost functions for our algebraic operators using regression analysis.

An overview of XQuery optimization in DB2 is given in [BEH+06]. The optimizations used there focus on the evaluation of path expressions. The compiler of Galax uses an algebra similar to ours to optimize XQuery [RSF06]. However, all optimizations in Galax are applied as heuristics without considering cost information.

As we have seen in this chapter, discarding order has great potential for optimizing XQuery. For example, more algebraic equivalences can be applied without destroying order. Grust et al. [GRT07] present a framework to derive all cases where order need not be preserved. Such cases include that the ordering mode is `unordered`, inside the `unordered` function, inside aggregates or quantifiers. These ideas are complementary to ours but support our claim that preserving order can be very costly.

Join Ordering

Join ordering is still the core of query optimization in relational databases. The fundamental work on join ordering was done in the System R prototype [SAC+79]. In general, the join ordering problem is NP-complete [OL90, PGLK97, Van98]. Recently, a join ordering algorithm was developed that generates query execution plans without cross products [NM06, Moe06]. This algorithm enumerates the minimal number of plan alternatives for all topologies of connected query graphs.

The special case of join ordering for order-preserving joins is discussed by Wu et al. [WPJ03] for structural joins and by Moerkotte [Moe03] for arbitrary order-preserving joins. The main result of this work is that join ordering can be done in $O(n^3)$ for a query containing n order-preserving joins.

Query optimizers also consider operators beyond natural joins. Proper scheduling of outerjoins, antijoins, and grouping can improve the quality of plans significantly. We plan to incorporate the ideas of Rao et al. [RLL+01] and Yan et al. [YL94] into our optimizer.

The optimization of nested queries in the plan generators was considered in [GRS05]. While this work does not remove nested query blocks, we would like to do even query unnesting inside the cost-based query optimizer.

Order is the most relevant physical property considered by query optimizers because order can be exploited by sort-based operator implementations, most importantly the merge join. Consequently, optimization of order has been investigated in [SSM96]. These ideas were extended to incorporate secondary sorting or grouping [WC03]. These techniques can be applied considering document order as one interesting order. [NM04] presents an efficient framework for order optimization on which we rely in our cost-based query optimizer.

Architectures for Query Optimizers

The architecture of our cost-based optimizer still follows the basic principles developed in System R [SAC+79]. In particular, we enumerate all query execution plans using dynamic programming. However, in the meantime extensibility has become mandatory to support new algebraic operators and algorithms for existing algebraic operators. The key technique developed for this purpose are rules which were introduced in the Starburst optimizer [Fre87, Loh88, LFL88, OL90] and Exodus [GD87] and its successors [GM93, McK93, Gra95].

Volcano [McK93], Cascades [Gra95], and Starburst [Loh88] use enforcers or glue operators to enforce properties. However, properties seem to be integrated into optimizers in an ad-hoc fashion.

We extend the work of Das [DB95] and develop an integrated framework to efficiently manage and derive logical and physical properties. We also identify the need to schedule enforcers that interfere. This architecture allows us to handle document order and sequence order in our plan generator. Moreover, as we foresee the need for more properties for effective XQuery optimization, we are able to handle new properties with minor changes to our code.

6. Conclusion

6.1. Summary

We have presented an algebraic framework for optimizing XQuery. This framework allows us to prove the correctness of algebraic equivalences. At the same time, these equivalences can be implemented efficiently, and experiments show that our unnesting equivalences improve the query execution times by orders of magnitude. Hence, these optimizations are important building blocks for efficient XQuery engines.

Our algebra, NAL, is the foundation for our optimizations. We formalize and prove algebraic properties of NAL. The notion of linearity allows us to check the reorderability of algebraic operators. We only have to check syntactic and some semantic conditions at query optimization time. Unfortunately, many equivalences known from algebras over sets or bags do not hold for our algebra over sequences. The operators in NAL can be implemented efficiently. We also enumerate the implementations for the operators in NAL. For example, we investigate efficient implementations for the binary grouping operator. This operator is a corner stone for the efficient execution of queries containing grouping operations.

We prepare XQuery statements for the translation by using normalization rewrites. Our translation function maps the resulting XQuery statements into algebraic expressions. We define the semantics of our query representation in terms of our algebra. For subsequent optimization steps, we annotate the translated query with type and cardinality information. The computation of all type information required for XQuery is implemented in our schema management component. Currently, we rely on Markov Tables to estimate the cardinalities of path expressions. In our experiments we show that generating a Markov Table for an XML document is more than 100 times faster than other XML synopses with similar estimation quality. Moreover, the Markov Table can be updated efficiently when the underlying data is updated. Our estimation framework is designed to support other XML synopsis structures without changing client code.

For many queries, the translation of XQuery statements introduces nested blocks. Nested queries can occur in three basic patterns: one for existentially quantified queries, one for universally quantified queries, and one for queries with implicit grouping. We call the last pattern implicit grouping because XQuery does not have a grouping construct yet. Thus, users need to state grouping with nested queries. We demonstrate that a naive evaluation of nested queries is very inefficient. This clearly motivates the need for unnesting nested queries. We enumerate unnesting equivalences and support rewrites that allow us to remove nested query blocks for almost every nested query. While query unnesting has been investigated before, our techniques include the following new aspects: (1) We are the first to formally treat unnesting for an algebra over sequences. (2) We include non-equality correlation predicates. (3) We unnest query blocks whose evaluation inherently depends on enclosing query blocks. For every basic pattern, we present a decision tree which selects an unnesting equivalence. Among all applicable equivalences, the selected one results in the most efficient unnested query. These decision trees formalize our unnesting strategy and are the basis for the implementation of our unnesting framework. Extensive experiments demonstrate that our unnesting techniques are implemented efficiently and that the resulting unnested queries are faster to evaluate, usually by orders of magnitude. This is possible because our query representation is designed for fast pattern matching. The correctness of our unnesting equivalences is validated in proofs.

After merging query blocks by unnesting nested queries, the cost-based query optimizer has more opportunities to schedule algebraic operators and select efficient implementations for them. As a novel challenge, the cost-based optimizer needs to respect duplicates *and* order as required by the query. We investigate path expressions where we have to preserve document order and joins which have to preserve sequence order. For both types of order, we argue that the query optimizer should consider to disregard order temporarily and repair it later in the query execution plan. Of course, the decision to sort or to preserve order should be based on costs. Since our physical algebra also works on sequences of tuples, we have a tight control over order. The property framework described in this thesis, in conjunction with an efficient management of interesting orders, allows us to do this efficiently.

6.2. Future Work

The results presented in this thesis can be extended in various ways.

Unified Query Representation There is no consensus on the best representation of XQuery and XPath statements. Since optimization is a key issue for the success of XQuery in the real world, a unified query representation similar to the algebra for relational databases would improve the common understanding of XQuery optimization. We believe that the formal treatment of our algebra over sequences is an important step into this direction. Since our algebra is an extension of the relational algebra, the transition to an algebra over sequences of tuples would be rather small.

Cost-Based Optimization Cost-based optimization of XQuery is still in its infancy. While we have provided evidence that cost-based optimization is a must for efficient XQuery processing, there are still several problems to solve. Among them, we discuss cost-based query unnesting and developing cost-functions for native XML query processing separately. Here, we note that current XQuery optimizers are rather XQuery translators equipped with several heuristics. Given that cost functions are known for all available query processing algorithms, we would like to include the following optimizations into the cost-based optimizer:

Index Selection Currently, most systems employ indices whenever they are eligible.

XML Views While the containment problem for path expressions is only tractable for a restricted subset of XPath, exploiting (materialized) views has a great potential in improving XQuery processing.

Diverse Operator Support Our research has shown that there will not be a single best evaluation technique even for path expressions. The superiority of certain evaluation techniques was demonstrated for specific use cases. Choosing the best algorithm for arbitrary queries is still an open problem.

Cost-based Query Unnesting As demonstrated in our experiments, in rare cases query unnesting does not improve query execution times. This is often the case when subexpressions are duplicated. These cases would benefit from an integration of query unnesting into the cost-based query optimizer.

Cost Functions for Native XML Query Processing To the best of our knowledge, establishing the execution costs for very basic operators has been solved only by XQuery engines that work on relational storage. For the bulk of evaluation techniques that exploit the native XML storage structures, no cost functions are known. Learning techniques

were proposed to capture the complexities of semistructured data in terms of I/O and CPU processing costs. But an analytical model, as it was developed for most other operator implementations, would be more desirable.

Benchmarking XMark is the dominant benchmark to asses the efficency of XQuery processors. However, this benchmark is restricted to a narrow subset of XQuery, and it operates only on a single large document. Instead, we need a widely accepted benchmark for XQuery that includes all important use cases of XQuery. MemBeR is a community-driven effort to accumulate benchmark queries, query generators, and document generators. We have submitted all queries presented in this thesis to this project as a nucleus of a benchmark for nested queries.

A. Proofs

A.1. Algebra

In this section we present the proofs of the algebraic equivalences over NAL.

A.1.1. Proof of Equivalence 2.1

$$e_1 \times (e_2 \times e_3) = (e_1 \times e_2) \times e_3$$

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$e_1 \times (e_2 \times e_3) = \epsilon = (e_1 \times e_2) \times e_3$$

Inductive Hypothesis: $e_1 \times (e_2 \times e_3) = (e_1 \times e_2) \times e_3$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & (e_1 \oplus t) \times (e_2 \times e_3) = ((e_1 \oplus t) \times e_2) \times e_3 \\ \Leftrightarrow \quad & (e_1 \times (e_2 \times e_3)) \oplus (t\overline{\times}(e_2 \times e_3)) = ((e_1 \times e_2) \times e_3) \oplus ((t\overline{\times}e_2) \times e_3) \end{aligned}$$

By inductive hypothesis we know that $(e_1 \times (e_2 \times e_3)) = ((e_1 \times e_2) \times e_3)$. Hence, it remains to be shown that $(t\overline{\times}(e_2 \times e_3)) = ((t\overline{\times}e_2) \times e_3)$. We prove this by induction over the length of e_2

Helper Proof

$$(t\overline{\times}(e_2 \times e_3)) = ((t\overline{\times}e_2) \times e_3)$$

Proof by Induction over the length of sequence e_2

Base Case: $e_2 = \epsilon$

$$t\overline{\times}(e_2 \times e_3) = \epsilon = (t\overline{\times}e_2) \times e_3$$

Inductive Hypothesis: $t\overline{\times}(e_2 \times e_3) = (t\overline{\times}e_2) \times e_3$

Inductive Step: $e_2 \rightarrow e_2 \oplus s$

$$
\begin{aligned}
& t\overline{\times}((e_2 \oplus s) \times e_3) = (t\overline{\times}(e_2 \oplus s)) \times e_3 \\
\Leftrightarrow \quad & t\overline{\times}((e_2 \times e_3) \oplus (s\overline{\times}e_3)) = ((t\overline{\times}e_2) \oplus (t \circ s)) \times e_3 \\
\Leftrightarrow \quad & (t\overline{\times}(e_2 \times e_3)) \oplus (t\overline{\times}(s\overline{\times}e_3)) = ((t\overline{\times}e_2) \times e_3) \oplus ((t \circ s) \times e_3)
\end{aligned}
$$

By inductive hypothesis we know that $(t\overline{\times}(e_2 \times e_3)) = ((t\overline{\times}e_2) \times e_3)$. Hence, it remains to be shown that $(t\overline{\times}(s\overline{\times}e_3)) = ((t \circ s) \times e_3)$.

$$
\begin{aligned}
\text{lhs} &= t\overline{\times}(s\overline{\times}e_3) \\
&= ((t \circ s)\overline{\times}e_3) \\
&= ((t \circ s) \times e_3) \\
&= \text{rhs}
\end{aligned}
$$

Note that t and s are single tuples. Thus, the order in which they are combined with e_3 does not matter. For the same reason we can replace $\overline{\times}$ by $\times$ in the last step.

This concludes the proof for Eqv. 2.1.

A.1.2. Proof of Equivalence 2.2

$$e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) = (e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3$$

The equivalence holds if $\mathcal{F}(p_i) \subset \mathcal{A}(e_i) \cup \mathcal{A}(e_{i+1})$ and the result of evaluation p_1 and p_2 does not depend on the position of the items fed into the predicates.

$$
\begin{aligned}
\text{lhs} &= e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) \\
&= e_1 \bowtie_{p_1} (\sigma_{p_2}(e_2 \times e_3)) \\
&= \sigma_{p_1}(e_1 \times (\sigma_{p_2}(e_2 \times e_3))) \\
&\overset{(2.14)}{=} \sigma_{p_1}(\sigma_{p_2}(e_1 \times (e_2 \times e_3))) \\
&\overset{(2.1)}{=} \sigma_{p_1}(\sigma_{p_2}((e_1 \times e_2) \times e_3)) \\
&= \sigma_{p_1}((e_1 \times e_2) \bowtie_{p_2} e_3) \\
&= (e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3 \\
&= \text{rhs}
\end{aligned}
$$

A.1.3. Proof of Equivalence 2.3

$$e_1 \ltimes_{p_1} (e_2 \ltimes_{p_2} e_3) = (e_1 \ltimes_{p_1} e_2) \ltimes_{p_2} e_3$$

The equivalence holds if $\mathcal{F}(p_1) \subset \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$, $\mathcal{F}(p_2) \subset \mathcal{A}(e_2) \cup \mathcal{A}(e_3)$. Additionally neither the result of p_1 nor the result of p_2 depends on the position of the tuples of either input sequence. We also require that p_2 must be strong w.r.t e_2.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$e_1 \ltimes_{p_1} (e_2 \ltimes_{p_2} e_3) = \epsilon = (e_1 \ltimes_{p_1} e_2) \ltimes_{p_2} e_3$$

Inductive Hypothesis: $e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) = (e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3$ for $|e_1| \geq 0$.

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$
\begin{aligned}
& (e_1 \oplus t) \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) = ((e_1 \oplus t) \bowtie_{p_1} e_2) \bowtie_{p_2} e_3 \\
\Leftrightarrow \quad & (e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3)) \oplus (t \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3)) = ((e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3) \oplus ((t \bowtie_{p_1} e_2) \bowtie_{p_2} e_3)
\end{aligned}
$$

By inductive hypothesis we know that $(e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3)) = ((e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3)$. Hence, it remains to be shown that $(t \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3)) = ((t \bowtie_{p_1} e_2) \bowtie_{p_2} e_3)$. We prove this by induction over the length of e_2

Helper Proof

$$t \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) = (t \bowtie_{p_1} e_2) \bowtie_{p_2} e_3$$

if $\mathcal{F}(p_1) \subset \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$, $\mathcal{F}(p_2) \subset \mathcal{A}(e_2) \cup \mathcal{A}(e_3)$ and p_2 is strong w.r.t e_2.

Proof by Induction over the length of sequence e_2

Base Case: $e_2 = \epsilon$

$$
\begin{aligned}
\text{lhs} \quad &= \quad t \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) \\
&\overset{p_2 \text{ strong}}{=} \quad t \circ \bot_{\mathcal{A}(e_2) \cup \mathcal{A}(e_3)} \\
&= \quad (t \bowtie_{p_1} e_2) \bowtie_{p_2} e_3 \\
&= \quad \text{rhs}
\end{aligned}
$$

Inductive Hypothesis: $t \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) = (t \bowtie_{p_1} e_2) \bowtie_{p_2} e_3$ for $|e_2| \geq 0$.

Inductive Step: $e_2 \rightarrow e_2 \oplus s$

$$
\begin{aligned}
& t \bowtie_{p_1} ((e_2 \oplus s) \bowtie_{p_2} e_3) = (t \bowtie_{p_1} (e_2 \oplus s)) \bowtie_{p_2} e_3 \\
\Leftrightarrow \quad & t \bowtie_{p_1} ((e_2 \bowtie_{p_2} e_3) \oplus (s \bowtie_{p_1} e_3)) = ((t \bowtie_{p_1} e_2) \oplus (t \bowtie_{p_1} s)) \bowtie_{p_2} e_3 \\
\Leftrightarrow \quad & (t \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3)) \oplus (t \bowtie_{p_1} (s \bowtie_{p_2} e_3)) = ((t \bowtie_{p_1} e_2) \bowtie_{p_2} e_3) \oplus ((t \bowtie_{p_1} s) \bowtie_{p_2} e_3)
\end{aligned}
$$

By inductive hypothesis we know that $(t \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3)) = ((t \bowtie_{p_1} e_2) \bowtie_{p_2} e_3)$. Hence, it remains to be shown that $(t \bowtie_{p_1} (s \bowtie_{p_2} e_3)) = ((t \bowtie_{p_1} s) \bowtie_{p_2} e_3)$.

Case 1: $s \bowtie_{p_2} e_3 \neq \epsilon$

Case 1.1: $p_1(t \circ s) = \text{true}$

$$
\begin{aligned}
\text{rhs} \quad &= \quad t \bowtie_{p_1} (s \bowtie_{p_2} e_3) \\
&= \quad t \bowtie_{p_1} (\sigma_{p_2}(s \overline{\times} e_3)) \\
&= \quad \sigma_{p_2}(t \bowtie_{p_1} (s \overline{\times} e_3)) \\
&= \quad \sigma_{p_2}((t \circ s) \overline{\times} e_3) \\
&= \quad (t \circ s) \bowtie_{p_2} e_3 \\
&= \quad (t \bowtie_{p_1} s) \bowtie_{p_2} e_3 \\
&= \quad \text{lhs}
\end{aligned}
$$

Case 1.2 $p_1(t \circ s) = \text{false}$

$$\begin{aligned}
\text{rhs} &= (t \bowtie_{p_1} s) \bowtie_{p_2} e_3 \\
&= (t \circ \bot_{\mathcal{A}(e_2)}) \bowtie_{p_2} e_3 \\
&\overset{p_2 \text{ strong}}{=} (t \circ \bot_{\mathcal{A}(e_2) \cup \mathcal{A}(e_3)}) \\
&= t \bowtie_{p_1} (s \bowtie_{p_2} e_3) \\
&= \text{lhs}
\end{aligned}$$

Case 2: $s \bowtie_{p_2} e_3 = \epsilon$

Case 2.1: $p_1(t \circ s) = \text{true}$

$$\begin{aligned}
\text{lhs} &= t \bowtie_{p_1} (s \bowtie_{p_2} e_3) \\
&= t \bowtie_{p_1} (s \circ \bot_{\mathcal{A}(e_3)}) \\
&= t \circ (s \circ \bot_{\mathcal{A}(e_3)}) \\
&= (t \circ s) \circ \bot_{\mathcal{A}(e_3)} \\
&= (t \bowtie_{p_1} s) \circ \bot_{\mathcal{A}(e_3)} \\
&= (t \bowtie_{p_1} s) \bowtie_{p_2} e_3 \\
&= \text{rhs}
\end{aligned}$$

Case 2.2 $p_1(t \circ s) = \text{false}$

$$\begin{aligned}
\text{lhs} &= t \bowtie_{p_1} (s \bowtie_{p_2} e_3) \\
&= t \bowtie_{p_1} (s \circ \bot_{\mathcal{A}(e_3)}) \\
&= t \circ \bot_{\mathcal{A}(e_2) \cup \mathcal{A}(e_3)} \\
&= (t \bowtie_{p_1} s) \bowtie_{p_2} e_3 \\
&= \text{rhs}
\end{aligned}$$

This concludes the proof for Eqv. 2.3.

A.1.4. Proof of Equivalence 2.4

$$e_1 \hat{\cup} (e_2 \hat{\cup} e_3) = (e_1 \hat{\cup} e_2) \hat{\cup} e_3$$

The equivalence holds if $\mathcal{A}(e_1) = \mathcal{A}(e_2) = \mathcal{A}(e_3)$.

The proof follows directly from the associativity of the sequence concatenation operator $\oplus$.

$$\begin{aligned}
\text{lhs} &= e_1 \hat{\cup} (e_2 \hat{\cup} e_3) \\
&= e_1 \oplus (e_2 \oplus e_3) \\
&= (e_1 \oplus e_2) \oplus e_3 \\
&= (e_1 \hat{\cup} e_2) \hat{\cup} e_3 \\
&= \text{rhs}
\end{aligned}$$

A.1.5. Proof of Equivalence 2.5

$$e_1 \hat{\cap} (e_2 \hat{\cap} e_3) = (e_1 \hat{\cap} e_2) \hat{\cap} e_3$$

The equivalence holds if $\mathcal{A}(e_1) = \mathcal{A}(e_2) = \mathcal{A}(e_3)$.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$e_1 \hat{\cap} (e_2 \hat{\cap} e_3) = \epsilon = (e_1 \hat{\cap} e_2) \hat{\cap} e_3$$

Inductive Hypothesis: $e_1 \hat{\cap} (e_2 \hat{\cap} e_3) = (e_1 \hat{\cap} e_2) \hat{\cap} e_3$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& (e_1 \oplus t) \hat{\cap} (e_2 \hat{\cap} e_3) = ((e_1 \oplus t) \hat{\cap} e_2) \hat{\cap} e_3 \\
\Leftrightarrow\ & (e_1 \oplus t) \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} (e_2 \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3) = ((e_1 \oplus t) \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2) \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3 \\
\Leftrightarrow\ & (e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} (e_2 \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3)) \oplus (t \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} (e_2 \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3)) \\
& \quad = ((e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2) \oplus (t \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2)) \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3 \\
& \quad = ((e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2) \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3) \oplus ((t \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2) \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3)
\end{aligned}$$

By inductive hypothesis we know that $(e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} (e_2 \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3)) = ((e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2) \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3)$ Hence, it remains to be shown that $(t \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} (e_2 \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3)) = ((t \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2) \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3)$.

$$\begin{aligned}
& t \in (t \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} (e_2 \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3)) \\
\Leftrightarrow\ & \exists x \in (e_2 \ltimes_{\mathcal{A}(e_2)=\mathcal{A}(e_3)} e_3) : t = x \\
\Leftrightarrow\ & \exists x \in e_2 : \exists y \in e_3 : t = x \wedge x = y \\
\Leftrightarrow\ & t = x = y \\
\Leftrightarrow\ & \exists x \in e_2 : t = x \wedge \exists y \in e_3 : t = y \\
\Leftrightarrow\ & (t \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_2)} e_2) \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_3)} e_3
\end{aligned}$$

A.1.6. Proof of Equivalence 2.6

$$\sigma_{p_1}(\sigma_{p_2}(e)) = \sigma_{p_2}(\sigma_{p_1}(e))$$

Proof by Induction over the length of sequence e

Base Case: $e = \epsilon$

$$\sigma_{p_1}(\sigma_{p_2}(e)) = \epsilon = \sigma_{p_2}(\sigma_{p_1}(e))$$

Inductive Hypothesis: $\sigma_{p_1}(\sigma_{p_2}(e)) = \sigma_{p_2}(\sigma_{p_1}(e))$, $|e_1| \geq 0$

Inductive Step: $e \rightarrow e \oplus t$

$$\begin{aligned}
& \sigma_{p_1}(\sigma_{p_2}(e \oplus t)) = \sigma_{p_2}(\sigma_{p_1}(e \oplus t)) \\
\Leftrightarrow\ & \sigma_{p_1}(\sigma_{p_2}(e)) \oplus \sigma_{p_1}(\sigma_{p_2}(t)) = \sigma_{p_2}(\sigma_{p_1}(e)) \oplus \sigma_{p_2}(\sigma_{p_1}(t))
\end{aligned}$$

By inductive hypothesis we know that $\sigma_{p_1}(\sigma_{p_2}(e)) = \sigma_{p_2}(\sigma_{p_1}(e))$. Thus, we need to show that $\sigma_{p_1}(\sigma_{p_2}(t)) = \sigma_{p_2}(\sigma_{p_1}(\alpha(t)))$:

case 1: $p_1(t)$ = true, then:

$$\begin{aligned}\sigma_{p_1}(\sigma_{p_2}(t)) &= \sigma_{p_2}(t)\\ &= \sigma_{p_2}(\sigma_{p_1}(t))\end{aligned}$$

case 2: $p_1(t)$ = false, then:

$$\begin{aligned}\sigma_{p_1}(\sigma_{p_2}(t)) &= \epsilon\\ &= \sigma_{p_2}(\sigma_{p_1}(t))\end{aligned}$$

Note that the result is the same only if neither the result of predicate p_1 nor the one of p_2 depends on the position of $\alpha(e)$ within the sequence computed by e. I.e. they may not be positional predicates.

A.1.7. Proof of Equivalence 2.7

$$\sigma_{p_1}(e_1 \times e_2) = \sigma_{p_1}(e_1) \times e_2$$

Note that this requires $\mathcal{F}(p_1) \subset \mathcal{A}(e_1)$.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$\sigma_{p_1}(e_1 \times e_2) = \epsilon = \sigma_{p_1}(e_1) \times e_2$$

Inductive Hypothesis: $\sigma_{p_1}(e_1 \times e_2) = \sigma_{p_1}(e_1) \times e_2$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}&\sigma_{p_1}((e_1 \oplus t) \times e_2) = \sigma_{p_1}(e_1 \oplus t) \times e_2\\ \Leftrightarrow\quad &\sigma_{p_1}(e_1 \times e_2) \oplus \sigma_{p_1}(t \overline{\times} e_2) = (\sigma_{p_1}(e_1) \times e_2) \oplus (\sigma_{p_1}(t) \overline{\times} e_2)\end{aligned}$$

By inductive hypothesis we know that $\sigma_{p_1}(e_1 \times e_2) = \sigma_{p_1}(e_1) \times e_2$. Thus, we need to show that $\sigma_{p_1}(t \overline{\times} e_2) = \sigma_{p_1}(t) \overline{\times} e_2$:

case 1: $p_1(t)$ = true, then:

$$\begin{aligned}\sigma_{p_1}(t \overline{\times} e_2) &= t \overline{\times} e_2\\ &= \sigma_{p_1}(t) \overline{\times} e_2\end{aligned}$$

case 2: $p_1(t)$ = false, then:

$$\begin{aligned}\sigma_{p_1}(t \overline{\times} e_2) &= \epsilon\\ &= \sigma_{p_1}(t) \overline{\times} e_2\end{aligned}$$

Note that the result is the same only if the result of predicate p_1 does not depend on the position of $\alpha(e)$ within the sequence computed by e_1. I.e. it may not be a positional predicate.

A.1.8. Proof of Equivalence 2.8

$$\sigma_{p_1}(e_1 \bowtie_p e_2) = \sigma_{p_1}(e_1) \bowtie_p e_2$$

This requires $\mathcal{F}(p_1) \subset \mathcal{A}(e_1)$ and neither predicate depends on the position of tuples in e_1.

$$\begin{aligned}
\text{lhs} &= \sigma_{p_1}(e_1 \bowtie_p e_2) \\
&= \sigma_{p_1}(\sigma_p(e_1 \times e_2)) \\
&\overset{(2.6)}{=} \sigma_p(\sigma_{p_1}(e_1 \times e_2)) \\
&\overset{(2.7)}{=} \sigma_p(\sigma_{p_1}(e_1) \times e_2) \\
&= \sigma_{p_1}(e_1) \bowtie_p e_2 \\
&= \text{rhs}
\end{aligned}$$

A.1.9. Proof of Equivalence 2.9

$$\sigma_{p_1}(e_1 \ltimes_p e_2) = \sigma_{p_1}(e_1) \ltimes_p e_2$$

This requires $\mathcal{F}(p_1) \subset \mathcal{A}(e_1)$ and neither predicate depends on the position of tuples in e_1.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$\sigma_{p_1}(e_1 \ltimes_p e_2) = \epsilon = \sigma_{p_1}(e_1) \ltimes_p e_2$$

Inductive Hypothesis: $\sigma_{p_1}(e_1 \ltimes_p e_2) = \sigma_{p_1}(e_1) \ltimes_p e_2$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& \sigma_{p_1}((e_1 \oplus t) \ltimes_p e_2) = \sigma_{p_1}(e_1 \oplus t) \ltimes_p e_2 \\
\Leftrightarrow\ & (\sigma_{p_1}(e_1 \ltimes_p e_2)) \oplus (\sigma_{p_1}(t \ltimes_p e_2)) = (\sigma_{p_1}(e_1) \ltimes_p e_2) \oplus (\sigma_{p_1}(t) \ltimes_p e_2)
\end{aligned}$$

By inductive hypothesis we know that $\sigma_{p_1}(e_1 \ltimes_p e_2) = (\sigma_{p_1}(e_1) \ltimes_p e_2)$. Hence, it remains to be shown that $(\sigma_{p_1}(t \ltimes_p e_2)) = (\sigma_{p_1}(t) \ltimes_p e_2)$.

case 1: $p_1(t)$ = true, then:

case 1.1: $\exists x \in e_2 : p(x \circ t)$ then

$$\text{lhs} = \sigma_{p_1}(t \ltimes_p e_2) = \sigma_{p_1}(t) = t = t \ltimes_p e_2 = \sigma_{p_1}(t) \ltimes_p e_2 = \text{rhs}$$

case 1.2: $\neg\exists x \in e_2 : p(x \circ t)$ then

$$\text{lhs} = \sigma_{p_1}(t \ltimes_p e_2) = \sigma_{p_1}(\epsilon) = \epsilon = t \ltimes_p e_2 = \sigma_{p_1}(t) \ltimes_p e_2 = \text{rhs}$$

case 2: $p_1(t)$ = false, then:

case 2.1: $\exists x \in e_2 : p(x \circ t)$ then

$$\text{lhs} = \sigma_{p_1}(t \ltimes_p e_2) = \sigma_{p_1}(t) = \epsilon = (\epsilon) \ltimes_p e_2 = \sigma_{p_1}(t) \ltimes_p e_2 = \text{rhs}$$

case 2.2: $\neg\exists x \in e_2 : p(x \circ t)$ then

$$\text{lhs} = \sigma_{p_1}(t \ltimes_p e_2) = \sigma_{p_1}(\epsilon) = \epsilon = (\epsilon) \ltimes_p e_2 = \sigma_{p_1}(t) \ltimes_p e_2 = \text{rhs}$$

A.1.10. Proof of Equivalence 2.10

$$\sigma_{p_1}(e_1 \triangleright_p e_2) = \sigma_{p_1}(e_1) \triangleright_p e_2$$

This requires $\mathcal{F}(p_1) \subset \mathcal{A}(e_1)$ and neither predicate depends on the position of tuples in e_1.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$\sigma_{p_1}(e_1 \triangleright_p e_2) = \epsilon = \sigma_{p_1}(e_1) \triangleright_p e_2$$

Inductive Hypothesis: $\sigma_{p_1}(e_1 \triangleright_p e_2) = \sigma_{p_1}(e_1) \triangleright_p e_2$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{p_1}((e_1 \oplus t) \triangleright_p e_2) = \sigma_{p_1}(e_1 \oplus t) \triangleright_p e_2 \\ \Leftrightarrow \quad & (\sigma_{p_1}(e_1 \triangleright_p e_2)) \oplus (\sigma_{p_1}(t \triangleright_p e_2)) = (\sigma_{p_1}(e_1) \triangleright_p e_2) \oplus (\sigma_{p_1}(t) \triangleright_p e_2) \end{aligned}$$

By inductive hypothesis we know that $\sigma_{p_1}(e_1 \triangleright_p e_2) = (\sigma_{p_1}(e_1) \triangleright_p e_2)$. Hence, it remains to be shown that $(\sigma_{p_1}(t \triangleright_p e_2)) = (\sigma_{p_1}(t) \triangleright_p e_2)$.

case 1: $p_1(t)$ = true, then:

case 1.1: $\exists x \in e_2 : p(x \circ t)$ then

$$\text{lhs} = \sigma_{p_1}(t \triangleright_p e_2) = \sigma_{p_1}(\epsilon) = \epsilon = t \triangleright_p e_2 = \sigma_{p_1}(t) \triangleright_p e_2 = \text{rhs}$$

case 1.2: $\neg\exists x \in e_2 : p(x \circ t)$ then

$$\text{lhs} = \sigma_{p_1}(t \triangleright_p e_2) = \sigma_{p_1}(t) = t = t \triangleright_p e_2 = \sigma_{p_1}(t) \triangleright_p e_2 = \text{rhs}$$

case 2: $p_1(t)$ = false, then:

case 2.1: $\exists x \in e_2 : p(x \circ t)$ then

$$\text{lhs} = \sigma_{p_1}(t \triangleright_p e_2) = \sigma_{p_1}(t) = \epsilon = (\epsilon) \triangleright_p e_2 = \sigma_{p_1}(t) \triangleright_p e_2 = \text{rhs}$$

case 2.2: $\neg\exists x \in e_2 : p(x \circ t)$ then

$$\text{lhs} = \sigma_{p_1}(t \triangleright_p e_2) = \sigma_{p_1}(t) = \epsilon = (\epsilon) \triangleright_p e_2 = \sigma_{p_1}(t) \triangleright_p e_2 = \text{rhs}$$

A.1.11. Proof of Equivalence 2.11

$$\sigma_{p_1}(e_1 \ltimes_p^{g:e} e_2) = \sigma_{p_1}(e_1) \ltimes_p^{g:e} e_2$$

This requires $\mathcal{F}(p_1) \subset \mathcal{A}(e_1)$ and neither predicate depends on the position of tuples in e_1.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$\sigma_{p_1}(e_1 \ltimes_p^{g:e} e_2) = \epsilon = \sigma_{p_1}(e_1) \ltimes_p^{g:e} e_2$$

Inductive Hypothesis: $\sigma_{p_1}(e_1 \ltimes_p^{g:e} e_2) = \sigma_{p_1}(e_1) \ltimes_p^{g:e} e_2$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{p_1}((e_1 \oplus t) \ltimes_p^{g:e} e_2) = \sigma_{p_1}(e_1 \oplus t) \ltimes_p^{g:e} e_2 \\ \Leftrightarrow \quad & (\sigma_{p_1}(e_1 \ltimes_p^{g:e} e_2)) \oplus (\sigma_{p_1}(t \ltimes_p^{g:e} e_2)) = (\sigma_{p_1}(e_1) \ltimes_p^{g:e} e_2) \oplus (\sigma_{p_1}(t) \ltimes_p^{g:e} e_2) \end{aligned}$$

By inductive hypothesis we know that $\sigma_{p_1}(e_1 \ltimes_p^{g:e} e_2) = (\sigma_{p_1}(e_1) \ltimes_p^{g:e} e_2)$. Hence, it remains to be shown that $(\sigma_{p_1}(t \ltimes_p^{g:e} e_2)) = (\sigma_{p_1}(t) \ltimes_p^{g:e} e_2)$.

case 1: $t \bowtie_p e_2 \neq \epsilon$, then:

$$\text{lhs} = \sigma_{p_1}(t \bowtie_p e_2) \overset{(2.8)}{=} \sigma_{p_1}(t) \bowtie_p e_2 = \text{rhs}$$

case 2: $t \bowtie_p e_2 = \epsilon$, then:

$$\text{lhs} = \sigma_{p_1}(t \circ \perp_{\mathcal{A}(e_2)\setminus\{g\}} \circ [g:e]) \overset{\mathcal{F}(p_1)\subset\mathcal{A}(e_1)}{=} \sigma_{p_1}(t) \perp_{\mathcal{A}(e_2)\setminus\{g\}} \circ [g:e] = \text{rhs}$$

A.1.12. Proof of Equivalence 2.12

$$\sigma_p(e_1 \Gamma_{g;A_1\theta A_2;f} e_2) = (\sigma_p(e_1))\Gamma_{g;A_1\theta A_2;f} e_2$$

with $\mathcal{F}(p) \subset \mathcal{A}(e_1), A_i \subset \mathcal{A}(e_i)$, and the result of evaluation predicate p does not depend on the position of items in its input sequence.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$:
$\text{lhs} = \epsilon = \text{rhs}$

Inductive Hypothesis: $\sigma_p(e_1 \Gamma_{g;A_1\theta A_2;f} e_2) = (\sigma_p(e_1))\Gamma_{g;A_1\theta A_2;f} e_2, |e_1| > 0$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& \sigma_p((e_1 \oplus t)\Gamma_{g;A_1\theta A_2;f} e_2) = (\sigma_p(e_1 \oplus t))\Gamma_{g;A_1\theta A_2;f} e_2 \\
\Leftrightarrow \quad & \sigma_p((e_1\Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)])) = (\sigma_p(e_1) \oplus \sigma_p(t))\Gamma_{g;A_1\theta A_2;f} e_2 \\
\Leftrightarrow \quad & \sigma_p(e_1\Gamma_{g;A_1\theta A_2;f} e_2) \oplus \sigma_p(t \circ [g : G(t)]) = (\sigma_p(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \oplus (\sigma_p(t)\Gamma_{g;A_1\theta A_2;f} e_2)
\end{aligned}$$

By inductive hypothesis we know that $\sigma_p(e_1\Gamma_{g;A_1\theta A_2;f} e_2) = (\sigma_p(e_1))\Gamma_{g;A_1\theta A_2;f} e_2$. Hence, it remains to be shown that $\sigma_p(t \circ [g : G(t)]) = \sigma_p(t)\Gamma_{g;A_1\theta A_2;f} e_2$:

case 1: $p(t) =$ true, then

$$\begin{aligned}
\text{lhs} &= \sigma_p(t \circ [g : G(t)]) \\
&= t \circ [g : G(t)] \\
&= t\Gamma_{g;A_1\theta A_2;f} e_2 \\
&= \text{rhs}
\end{aligned}$$

case 2: $p(t) =$ false, then

$$\begin{aligned}
\text{lhs} &= \sigma_p(t \circ [g : G(t)]) \\
&= \epsilon \\
&= \epsilon\Gamma_{g;A_1\theta A_2;f} e_2 \\
&= \text{rhs}
\end{aligned}$$

A.1.13. Proof of Equivalence 2.13

$$\sigma_p(e_1 \hat{\cup} e_2) = \sigma_p(e_1) \hat{\cup} \sigma_p(e_2)$$

We prove this equivalence in two inductive proofs. The first is an induction over the length of sequence e_1, the second over length of sequence e_2.

1. Proof by Induction over the length of sequence e_1 Let e_2 be a sequence of arbitrary but fixed length.

Base Case: $e_1 = \epsilon$

$$\sigma_p(e_1 \hat{\cup} e_2) = \sigma_p(e_2) = \sigma_p(e_1) \hat{\cup} e_2$$

Inductive Hypothesis: $\sigma_p(e_1 \hat{\cup} e_2) = \sigma_p(e_1) \hat{\cup} \sigma_p(e_2), |e_1| ge 0$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\sigma_p((e_1 \oplus t) \hat{\cup} e_2) = \sigma_p((e_1 \oplus t)) \hat{\cup} \sigma_p(e_2)$$

case 1: $p(\alpha(e_1))$ = true, then the lhs evaluates to:

$$\sigma_p((e_1 \oplus t) \hat{\cup} e_2) = \alpha(e_1) \oplus \sigma_p(\tau(e_1 \oplus t) \hat{\cup} e_2)$$

The rhs evaluates to:

$$\sigma_p((e_1 \oplus t)) \hat{\cup} e_2) = \alpha(e_1) \oplus \sigma_p(\tau(e_1 \oplus t)) \hat{\cup} e_2$$

By inductive hyposthesis we know that $\sigma_p(\tau(e_1 \oplus t) \hat{\cup} e_2) = \sigma_p(\tau(e_1 \oplus t)) \hat{\cup} e_2$. Hence, both sides are the same.

case 2: $p_1(\alpha(e_1))$ = false, then $\alpha(e_1)$ will not be in the result. Hence we get: lhs $=$ $\sigma_p(\tau(e_1 \oplus t)) \hat{\cup} e_2 =$ rhs

2. Proof by Induction over the length of sequence e_2 Let e_1 be a sequence of arbitrary but fixed length.

Base Case: $e_2 = \epsilon$

$$\sigma_p(e_1 \hat{\cup} e_2) = \sigma_p(e_1) = \sigma_p(e_1) \hat{\cup} e_2$$

Inductive Hypothesis: $\sigma_p(e_1 \hat{\cup} e_2) = \sigma_p(e_1) \hat{\cup} \sigma_p(e_2), |e_1| \geq 0$

Inductive Step: $e_2 \rightarrow e_2 \oplus s$

$$\begin{aligned}
\text{lhs} &= \sigma_p(e_1 \hat{\cup} (e_2 \oplus s)) \\
&= \sigma_p(e_1 \oplus (e_2 \oplus s)) \\
&= \sigma_p((e_1 \oplus e_2) \oplus s) \\
&= \sigma_p(e_1 \oplus e_2) \oplus \sigma_p(s) \\
&= \sigma_p(e_1 \hat{\cup} e_2) \oplus \sigma_p(s) \\
&\stackrel{*}{=} \sigma_p(e_1) \hat{\cup} \sigma_p(e_2) \oplus \sigma_p(s) \\
&= \sigma_p(e_1) \hat{\cup} \sigma_p(e_2 \oplus s) \\
&= \text{rhs}
\end{aligned}$$

A.1.14. Proof of Equivalence 2.14

$$\sigma_{p_2}(e_1 \times e_2) = e_1 \times \sigma_{p_2}(e_2)$$

Note that this requires $\mathcal{F}(p_2) \subset \mathcal{A}(e_2)$.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$\sigma_{p_2}(e_1 \times e_2) = \epsilon = e_1 \times \sigma_{p_2}(e_2)$$

Inductive Hypothesis: $\sigma_{p_2}(e_1 \times e_2) = e_1 \times \sigma_{p_2}(e_2)$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& \sigma_{p_2}((e_1 \oplus t) \times e_2) = (e_1 \oplus t) \times \sigma_{p_2}(e_2) \\
\Leftrightarrow\ & \sigma_{p_2}(e_1 \times e_2) \oplus \sigma_{p_2}(t \overline{\times} e_2) = (e_1 \times \sigma_{p_2}(e_2)) \oplus (t \overline{\times} \sigma_{p_2}(e_2))
\end{aligned}$$

By inductive hypothesis we know that $\sigma_{p_2}(e_1 \times e_2) = e_1 \times \sigma_{p_2}(e_2)$. Thus, we need to show that $\sigma_{p_2}(t \overline{\times} e_2) = t \overline{\times} \sigma_{p_2}(e_2)$.

Helper Proof

$$\sigma_{p_2}(t \overline{\times} e_2) = t \overline{\times} \sigma_{p_2}(e_2)$$

with $\mathcal{F}(p_2) \subset \mathcal{A}(e_2)$.

Proof by Induction over the length of sequence e_2

Base Case: $e_2 = \epsilon$

$$\sigma_{p_2}(t\overline{\times}e_2) = \epsilon = t\overline{\times}\sigma_{p_2}(e_2)$$

Inductive Hypothesis: $\sigma_{p_2}(t\overline{\times}e_2) = t\overline{\times}\sigma_{p_2}(e_2)$

Inductive Step: $e_2 \rightarrow e_2 \oplus s$

$$\begin{aligned} & \sigma_{p_2}(t\overline{\times}(e_2 \oplus s)) = t\overline{\times}\sigma_{p_2}((e_2 \oplus s)) \\ \Leftrightarrow \quad & \sigma_{p_2}(t\overline{\times}e_2) \oplus \sigma_{p_2}(t \circ s) = (t\overline{\times}\sigma_{p_2}(e_2)) \oplus (t \circ \sigma_{p_2}(s)) \end{aligned}$$

By inductive hyposthesis we know $\sigma_{p_2}(t\overline{\times}e_2) = t\overline{\times}\sigma_{p_2}(e_2)$. Hence, we need to show that $\sigma_{p_2}(t \circ s) = t \circ \sigma_{p_2}(s)$

case 1: $p_2(s)$ = true, then:

$$\begin{aligned} \sigma_{p_2}(t \circ s) &= t \circ s \\ &= t \circ \sigma_{p_2}(s) \end{aligned}$$

case 2: $p_2(s)$ = false, then:

$$\begin{aligned} \sigma_{p_2}(t \circ s) &= \epsilon \\ &= t \circ \sigma_{p_2}(s) \end{aligned}$$

This poofs the helper proof.

As a result Eqv. 2.14 holds, when the result of predicate p_2 does not depend on the position of $\alpha(e_2)$ within the sequence computed by e_2. I.e. it may not be a positional predicate.

A.1.15. Proof of Equivalence 2.15

$$\sigma_{p_2}(e_1 \bowtie_p e_2) = e_1 \bowtie_p \sigma_{p_2}(e_2)$$

This requires $\mathcal{F}(p_2) \subset \mathcal{A}(e_2)$ and neither predicate depends on the position of tuples in e_2.

$$\begin{aligned} \text{lhs} &= \sigma_{p_2}(e_1 \bowtie_p e_2) \\ &= \sigma_{p_2}(\sigma_p(e_1 \times e_2)) \\ &\overset{(2.6)}{=} \sigma_p(\sigma_{p_2}(e_1 \times e_2)) \\ &\overset{(2.14)}{=} \sigma_p(e_1 \times \sigma_{p_2}(e_2)) \\ &= e_1 \bowtie_p \sigma_{p_2}(e_2) \\ &= \text{rhs} \end{aligned}$$

A.1.16. Proof of Equivalence 2.16

$$\Pi_A(\Pi_B(e_1)) = \Pi_A(e_1)$$

if $B \subset A$

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$\Pi_A(\Pi_B(e_1)) = \epsilon = \Pi_A(e_1)$$

Inductive Hypothesis: $\Pi_A(\Pi_B(e_1)) = \Pi_A(e_1)$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \Pi_A(\Pi_B(e_1 \oplus t)) = \Pi_A(e_1 \oplus t) \\ \Leftrightarrow \quad & \Pi_A(\Pi_B(e_1)) \oplus \alpha(\Pi_B(t)_{|A}) = \Pi_A(e_1) \oplus \alpha(t)_{|A} \end{aligned}$$

By inductive hypothesis we know that $\Pi_A(\Pi_B(e_1)) = \Pi_A(e_1)$. Hence, it remains to be shown that $\alpha(\Pi_B(t)_{|A}) = \alpha(t)_{|A}$.

$$\begin{aligned} \text{lhs} &= \alpha(\Pi_B(t)_{|A}) \\ &= \alpha((\alpha(t)_{|B})_{|A}) \\ &= \alpha(t)_{|A} \\ &= \text{rhs} \end{aligned}$$

A.1.17. Proof of Equivalence 2.17

$$\Upsilon_{\mathcal{A}(e_2):e_2}(e_1) = e_1 \underrightarrow{\bowtie} e_2$$

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$\Upsilon_{\mathcal{A}(e_2):e_2}(e_1) = \epsilon = e_1 \underrightarrow{\bowtie} e_2$$

Inductive Hypothesis: $\Upsilon_{\mathcal{A}(e_2):e_2}(e_1) = e_1 \underrightarrow{\bowtie} e_2$ for $|e_1| \geq 0$.

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \Upsilon_{\mathcal{A}(e_2):e_2}(e_1 \oplus t) = (e_1 \oplus t) \underrightarrow{\bowtie} e_2 \\ \Leftrightarrow \quad & \mu_{\mathcal{A}(e_2):\hat{a}}(\chi_{\hat{a}:e_2}(e_1 \oplus t)) = (e_1 \oplus t) \underrightarrow{\bowtie} e_2 \\ \Leftrightarrow \quad & \mu_{\mathcal{A}(e_2):\hat{a}}(\chi_{\hat{a}:e_2}(e_1)) \oplus \mu_{\mathcal{A}(e_2):\hat{a}}(t \circ [\hat{a} : e_2(t)]) = (e_1 \underrightarrow{\bowtie} e_2) \oplus (t \underrightarrow{\bowtie} e_2) \\ \Leftrightarrow \quad & \Upsilon_{\mathcal{A}(e_2):e_2}(e_1) \oplus \mu_{\mathcal{A}(e_2):\hat{a}}(t \circ [\hat{a} : e_2(t)]) = (e_1 \underrightarrow{\bowtie} e_2) \oplus (t \underrightarrow{\bowtie} e_2) \end{aligned}$$

By inductive hypothesis we know that $\Upsilon_{\mathcal{A}(e_2):e_2}(e_1) = e_1 \underrightarrow{\bowtie} e_2$. Hence, it remains to be shown that $\mu_{\mathcal{A}(e_2):\hat{a}}(t \circ [\hat{a} : e_2(t)]) = t \underrightarrow{\bowtie} e_2$.

$$\begin{aligned}
\text{lhs} &= \mu_{\mathcal{A}(e_2):\hat{a}}(t \circ [\hat{a} : e_2(t)]) \\
&= t \times (\Pi_{\mathcal{A}(e_2):\mathcal{A}(e_2)}(e_2(t))) \\
&= t \overline{\times} e_2(t) \\
&= t \underrightarrow{\bowtie} e_2 \\
&= \text{rhs}
\end{aligned}$$

In the third step marked, we can throw away the sequence-valued attribute $\hat{a}$ because it is only used temporarily inside the Υ operator.

A.1.18. Proof of Equivalence 2.18

$$e_1 \underrightarrow{\bowtie} e_2 = e_1 \times e_2$$

The proof is a trivial consequence of the requirement that $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, i.e. the result of evaluating e_2 does not dependent on e_1.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$

$$e_1 \underrightarrow{\bowtie} e_2 = \epsilon = e_1 \times e_2$$

Inductive Hypothesis: $e_1 \underrightarrow{\bowtie} e_2 = e_1 \times e_2$ for $|e_1| \geq 0$.

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
\text{lhs} &= (e_1 \oplus t) \underrightarrow{\bowtie} e_2 \\
&= (e_1 \underrightarrow{\bowtie} e_2) \oplus (t \overline{\times} e_2(t)) \\
&= (e_1 \underrightarrow{\bowtie} e_2) \oplus (t \overline{\times} e_2) \\
&\overset{IH}{=} (e_1 \times e_2) \oplus (t \overline{\times} e_2) \\
&= (e_1 \oplus t) \times e_2 \\
&= \text{rhs}
\end{aligned}$$

In the third step we exploit the fact that the e_2 can be evaluated independent of e_1.

A.1.19. Proof of Equivalence 2.19

$$e_1 \Gamma_{g;A_1 \theta A_2;f} e_2 = \hat{\cup}_i (e_{1_i} \Gamma_{g;A_1 \theta A_2;f} e_2)$$

with $A_j \subset \mathcal{A}(e_j)$

Note that it suffices to show that this equivalence holds for two partitions where the first partition contains exactly one element, i.e. we will use partitions $\alpha(e_1)$ and $\tau(e_1)$. The case for arbitrary numbers of partitions and partition sizes follows directly from Eqv. 2.4.

Proof by Induction over the length of sequence e_1

Base Case:

$|e_1| = 0$: $e_1 \Gamma_{g;A_1\theta A_2;f} e_2 = \epsilon = \hat{\cup}_i (e_{1_i} \Gamma_{g;A_1\theta A_2;f} e_2)$

$|e_1| = 1$: We define $e_1 := t \,\hat{\cup}\, \epsilon$, then

$$\begin{aligned} \text{rhs} &= (t\, \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\epsilon\, \Gamma_{g;A_1\theta A_2;f} e_2) \\ &= (t\, \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, \epsilon \\ &= t\, \Gamma_{g;A_1\theta A_2;f} e_2 \\ &= \text{lhs} \end{aligned}$$

Inductive Hypothesis: $e_1 \Gamma_{g;A_1\theta A_2;f} e_2 = (\alpha(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\tau(e_1) \Gamma_{g;A_1\theta A_2;f} e_2)$, $|e_1| > 0$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

Note that in the following steps it suffices to write $\alpha(e_1)$ instead of $\alpha(e_1 \oplus t)$ because sequence e_1 is not empty ($|e_1| > 0$) and only the first izem of the sequence is retrieved. Hence $\alpha(e_1) = \alpha(e_1 \oplus t)$ holds.

$$\begin{aligned} & (e_1 \oplus t) \Gamma_{g;A_1\theta A_2;f} e_2 = (\alpha(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\tau(e_1 \oplus t) \Gamma_{g;A_1\theta A_2;f} e_2) \\ \Leftrightarrow\ & (e_1 \oplus t) \Gamma_{g;A_1\theta A_2;f} e_2 = (\alpha(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\tau(e_1 \oplus t) \Gamma_{g;A_1\theta A_2;f} e_2) \\ \Leftrightarrow\ & (e_1 \Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)]) = (\alpha(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, ((\tau(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)])) \\ \Leftrightarrow\ & (e_1 \Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)]) = (\alpha(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, ((\tau(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)])) \end{aligned}$$

In these steps $G(t) := f(\sigma_{t|_{A_1}\theta A_2}(e_2))$ as used in the definition of the binary grouping operator.

By inductive hypothesis we know that $e_1 \Gamma_{g;A_1\theta A_2;f} e_2 = (\alpha(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\tau(e_1) \Gamma_{g;A_1\theta A_2;f} e_2)$. Since the remaining expression is obviously equal, we have proven the equivalence.

A.1.20. Proof of Equivalence 2.19

$$e_1 \Gamma_{g;A_1\theta A_2;f} e_2 = \hat{\cup}_i (e_{1_i} \Gamma_{g;A_1\theta A_2;f} e_2)$$

with $A_j \subset \mathcal{A}(e_j)$

Note that it suffices to show that this equivalence holds for two partitions where the first partition contains exactly one element, i.e. we will use partitions $\alpha(e_1)$ and $\tau(e_1)$. The case for arbitrary numbers of partitions and partition sizes follows directly from Eqv. 2.4.

Proof by Induction over the length of sequence e_1

Base Case:

$|e_1| = 0$: $e_1 \Gamma_{g;A_1\theta A_2;f} e_2 = \epsilon = \hat{\cup}_i (e_{1_i} \Gamma_{g;A_1\theta A_2;f} e_2)$

$|e_1| = 1$: We define $e_1 := t \,\hat{\cup}\, \epsilon$, then

$$\begin{aligned} \text{rhs} &= (t\, \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\epsilon\, \Gamma_{g;A_1\theta A_2;f} e_2) \\ &= (t\, \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, \epsilon \\ &= t\, \Gamma_{g;A_1\theta A_2;f} e_2 \\ &= \text{lhs} \end{aligned}$$

Inductive Hypothesis: $e_1 \Gamma_{g;A_1\theta A_2;f} e_2 = (\alpha(e_1) \Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\tau(e_1) \Gamma_{g;A_1\theta A_2;f} e_2)$, $|e_1| > 0$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

Note that in the following steps it suffices to write $\alpha(e_1)$ instead of $\alpha(e_1 \oplus t)$ because sequence e_1 is not empty ($|e_1| > 0$) and only the first izem of the sequence is retrieved. Hence $\alpha(e_1) = \alpha(e_1 \oplus t)$ holds.

$$
\begin{aligned}
& (e_1 \oplus t)\Gamma_{g;A_1\theta A_2;f} e_2 = (\alpha(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\tau(e_1 \oplus t)\Gamma_{g;A_1\theta A_2;f} e_2) \\
\Leftrightarrow \quad & (e_1 \oplus t)\Gamma_{g;A_1\theta A_2;f} e_2 = (\alpha(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\tau(e_1 \oplus t)\Gamma_{g;A_1\theta A_2;f} e_2) \\
\Leftrightarrow \quad & (e_1\Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)]) = (\alpha(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, ((\tau(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)])) \\
\Leftrightarrow \quad & (e_1\Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)]) = (\alpha(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, ((\tau(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \oplus (t \circ [g : G(t)]))
\end{aligned}
$$

In these steps $G(t) := f(\sigma_{t|_{A_1}\theta A_2}(e_2))$ as used in the definition of the binary grouping operator.

By inductive hypothesis we know that $e_1\Gamma_{g;A_1\theta A_2;f} e_2 = (\alpha(e_1)\Gamma_{g;A_1\theta A_2;f} e_2) \,\hat{\cup}\, (\tau(e_1)\Gamma_{g;A_1\theta A_2;f} e_2)$. Since the remaining expression is obviously equal, we have proven the equivalence.

A.1.21. Proof of Equivalence 2.20

$$(e_1\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3 = (e_1\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2$$

with $\mathcal{F}(f_i) \subset \mathcal{A}(e_1) \cup \mathcal{A}(e_{i+1}), A_{1_i} \subset \mathcal{A}(e_1)$, and $A_j \subset \mathcal{A}(e_j), g_1 \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2), g_2 \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_3)$.

Proof by Induction over the length of sequence e_1

Base Case: $e_1 = \epsilon$:

lhs $= \epsilon =$ rhs

Inductive Hypothesis: $(e_1\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3 = (e_1\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2$, $|e_1| > 0$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$
\begin{aligned}
& ((e_1 \oplus t)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3 = ((e_1 \oplus t)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2 \\
\Leftrightarrow \quad & ((e_1\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2) \oplus (t\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2))\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3 \\
& = ((e_1\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3) \oplus (t\Gamma_{g_1;A_1\theta_2A_3;f_2} e_3))\Gamma_{g_2;A_1\theta_1A_2;f_1} e_2 \\
\Leftrightarrow \quad & ((e_1\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3) \oplus ((t\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3) \\
& = ((e_1\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2) \oplus ((t\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)
\end{aligned}
$$

By inductive hypothesis we know that $(e_1\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3 = (e_1\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2$. Hence, it remains to be shown that $(t\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3 = (t\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2$:

$$
\begin{aligned}
\text{lhs} &= (t\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2)\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3 \\
&= (t \circ [g_1 : G_1(t)])\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3 \\
&= ((t \circ [g_1 : G_1(t)]) \circ [g_2 : G_2(t)]) \\
&= ((t \circ [g_2 : G_2(t)]) \circ [g_1 : G_1(t)]) \\
&= (t \circ [g_2 : G_2(t)])\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2 \\
&= (t\Gamma_{g_2;A_1\theta_2A_3;f_2} e_3)\Gamma_{g_1;A_1\theta_1A_2;f_1} e_2 \\
&= \text{rhs}
\end{aligned}
$$

where $G_1(t) := f_1(\sigma_{t|_{A_1}\theta A_2}(e_2))$ and $G_2(t) := f_2(\sigma_{t|_{A_1}\theta A_3}(e_3))$.

A.1.22. Proof of Equivalence 2.21

$$e_1 \Gamma_{g;A_1=A_2;f} e_2 = \Pi_{\overline{A_2}}(e_1 \bowtie^{g:f(\epsilon)}_{A_1=A_2} (\Gamma_{g;=A_2;f}(e_2)))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $A_1 \cap A_2 = \emptyset$, and $g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$.

Case 1: $e_1 = \epsilon$

lhs $= \epsilon =$ rhs

Case 2: $e_1 \neq \epsilon$

Let t_i be the i-th tuple in e_1 and

$$\begin{aligned} h(e_2) &= \Gamma_{g;=A_2;f}(e_2) \\ &= \Pi_{A_2:A_2'}(\Pi^D_{A_2':A_2}(\Pi_{A_2}(e_2))\Gamma_{g;A_2'=A_2;f} e_2). \end{aligned}$$

e_2 is projected on A_2 with a duplicate elimination, so each value of A_2 appears only once in $h(e_2)$. Let t'_j be the j-th tuple in $\Pi^D_{A_2':A_2}(\Pi_{A_2}(e_2))$. The j-th tuple in $h(e_2)$ then is

$$\begin{aligned} & \Pi_{A_2:A_2'}(t'_j \circ [g : f(\sigma_{t_j|_{A_2'}=A_2}(e_2))]) \\ = \; & \Pi_{A_2:A_2'}(t'_j \circ [g : f(\sigma_{A_2'=A_2}(e_2))(t'_j)]). \end{aligned}$$

Each tuple t_i in e_1 joins with at most one tuple in $h(e_2)$ with join predicate $A_1 = A_2$. If no join partner is found in $h(e_2)$, then the current tuple of e_1 is padded with appropriate values. For each tuple t_i in e_1 we have the corresponding tuple at the i-th position after the outer join.

Case 2(a): $\neg\exists x \in e_2 : t_i.A_1 = x.A_2$
($\Rightarrow t_i \bowtie_{A_1=A_2} h(e_2) = \epsilon$)

For lhs we have

$$\begin{aligned} & t_i \circ [g : f(\sigma_{t_{i|A_1}=A_2}(e_2))] \\ = \; & t_i \circ [g : f(\epsilon)]. \end{aligned}$$

For the right hand side (rhs) we get

$$\begin{aligned} & \Pi_{\overline{A_2}}(t_i \circ \perp_{A_2} \circ [g : f(\epsilon)]) \\ = \; & t_i \circ [g : f(\epsilon)]. \end{aligned}$$

Case 2(b): $\exists x \in e_2 : t_i.A_1 = x.A_2$
($\Rightarrow t_i \bowtie_{A_1=A_2} h(e_2) \neq \epsilon$)

For the left hand side (lhs) we have

$$t_i \circ [g : f(\sigma_{t_{i|A_1}=A_2}(e_2))].$$

We now turn to rhs. Let t''_k be the tuple from $h(e_2)$ for which $t''_k.A_2 = t_i.A_1$ (all other tuples in $h(e_2)$ are irrelevant for the join). Therefore, rhs is equal to

$$\begin{aligned} & \Pi_{\overline{A_2}}(t_i \bowtie_{A_1=A_2} h(e_2)) \\ = \; & \Pi_{\overline{A_2}}(t_i \circ t''_k) \\ = \; & \Pi_{\overline{A_2}}(t_i \circ \Pi_{A_2:A'_2}(t'_k \circ [g : f(\sigma_{|t'_{k\,|A'_2}=A_2}(e_2))])). \end{aligned}$$

As $t_i.A_1 = t''_k.A_2 = t'_k.A'_2$ and we project away A'_2 (after renaming it to A_2), we get

$$t_i \circ [g : f(\sigma_{t_{i\,|A_1}=A_2}(e_2))].$$

A.2. Unnesting Equivalences

For completeness, we present the proofs of our unnesting equivalences. These proofs were done by Sven Helmer and have already appeared in [MHM06].

For the following proofs let lhs denote the left hand side and rhs the right hand side of an equivalence.

A.2.1. Proof of Equivalence 4.1

$$\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid_{i1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(tid_{i_1}(e_1))))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$,
Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:
lhs = rhs = ϵ

Inductive Hypothesis:
$\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(tid_{i_1}(e_1))))$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\exists x \in (e_2):p}(e_1 \oplus t) = \\ & \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(tid_{i_1}(e_1 \oplus t)))) \\ \Leftrightarrow \; & \sigma_{\exists x \in (e_2):p}(e_1) \oplus \sigma_{\exists x \in (e_2):p}(t) = \\ & \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(tid_{i_1}(e_1)))) \oplus \\ & \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(tid_{i_1}(t \circ [i_1 : \max(\Pi_{tid}(e_1)) + 1])))) \end{aligned}$$

As we know that $\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(tid_{i_1}(e_1))))$,
we have to prove that
$\sigma_{\exists x \in (e_2(t)):p}(t) = \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2(t)}(tid_{i_1}(t \circ [i_1 : \max(\Pi_{tid}(e_1)) + 1]))))$

Case 1: $\exists x \in e_2(t) : (p)(t \circ x)$
For the lhs, this means that t will pass the selection operator, so
$\sigma_{\exists x \in (e_2(t)):p}(t) = t$
For the rhs this means that $\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2(t)}(tid_{i_1}(t \circ [i_1 : \max(\Pi_{tid}(e_1)) + 1])))$ will contain all tuples in $\Upsilon_{\mathcal{A}(e_2):e_2(t)}(tid_{i_1}(t \circ [i_1 : \max(\Pi_{tid}(e_1)) + 1]))$ for which p holds, among them the tuple $(t \circ [i_1 : \max(\Pi_{tid}(e_1)) + 1]) \circ x$. As all tuples have the same attribute values for $\mathcal{A}(e_1)$, including i_1, the projection operator will reduce this to a single tuple, t.

Case 2: $\nexists x \in e_2(t) : (p)(t \circ x)$
For the lhs, this means that $\sigma_{\exists x \in (e_2(t)):p}(t) = \epsilon$
For the rhs, $\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2(t)}(tid_{i_1}(t \circ [i_1 : \max(\Pi_{tid}(e_1))+1])))$ will be empty, therefore rhs=ϵ.

A.2.2. Proof of Equivalence 4.2

$$\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid}_{\mathcal{A}(e1)}(\sigma_p(tid(e_1) \times e_2))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$,
Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:
lhs = rhs = ϵ

Inductive Hypothesis:
$\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid}_{\mathcal{A}(e1)}(\sigma_p(tid(e_1) \times e_2))$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\exists x \in (e_2):p}(e_1 \oplus t) = \\ & \Pi^{tid}_{\mathcal{A}(e1)}(\sigma_p(tid(e_1 \oplus t) \times e_2)) \\ \Leftrightarrow \quad & \sigma_{\exists x \in (e_2):p}(e_1) \oplus \sigma_{\exists x \in (e_2):p}(t) = \\ & \Pi^{tid}_{\mathcal{A}(e1)}(\sigma_p(tid(e_1) \times e_2)) \oplus \Pi^{tid}_{\mathcal{A}(e1)}(\sigma_p((t \circ [tid : \max(\Pi_{tid}(e_1)) + 1]) \times e_2)) \end{aligned}$$

As we know that $\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid}_{\mathcal{A}(e1)}(\sigma_p(tid(e_1) \times e_2))$, we have to prove that $\sigma_{\exists x \in (e_2):p}(t) = \Pi^{tid}_{\mathcal{A}(e1)}(\sigma_p((t \circ [tid : \max(\Pi_{tid}(e_1)) + 1]) \times e_2))$

Case 1: $\exists x \in e_2 : (p)(t \circ x)$
For the lhs, this means that t will pass the selection operator, so
$\sigma_{\exists x \in (e_2):p}(t) = t$
For the rhs this means that $\sigma_p((t \circ [tid : \max(\Pi_{tid}(e_1))+1]) \times e_2)$ will contain all tuples in $(t \circ [tid : \max(\Pi_{tid}(e_1)) + 1]) \times e_2$ for which p holds, among them the tuple $(t \circ [tid : \max(\Pi_{tid}(e_1)) + 1]) \circ x$. As all tuples have the same attribute values for $\mathcal{A}(e_1)$, including tid, the projection operator will reduce this to a single tuple, t.

Case 2: $\nexists x \in e_2 : (p)(t \circ x)$
For the lhs, this means that $\sigma_{\exists x \in (e_2):p}(t) = \epsilon$.
For the rhs, $\sigma_p((t \circ [tid : \max(\Pi_{tid}(e_1)) + 1]) \times e_2)$ will be empty, therefore rhs=ϵ.

A.2.3. Proof of Equivalence 4.3

$$\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1=A_2 \wedge p} e_2$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$.
Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:
lhs = rhs = ϵ

Inductive Hypothesis:
$\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1=A_2 \wedge p} e_2$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1 \oplus t) = \\ & (e_1 \oplus t) \ltimes_{A_1=A_2 \wedge p} e_2 \\ \Leftrightarrow \quad & \sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) \oplus \sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(t) = \\ & e_1 \ltimes_{A_1=A_2 \wedge p} e_2 \oplus t \ltimes_{A_1=A_2 \wedge p} e_2 \end{aligned}$$

As we know that $\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1=A_2 \wedge p} e_2$, we have to prove that $\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(t) = t \ltimes_{A_1=A_2 \wedge p} e_2$.

Case 1: $\exists x \in e_2 : (A_1 = A_2 \wedge p)(t \circ x)$

First of all, we show that $\exists x \in e_2 : (A_1 = A_2 \wedge p)(t \circ x) \Leftrightarrow \exists x \in (\sigma_{A_1=A_2}(e_2))(t) : p$.

"⇒":

Let y be a tuple from e_2 for which $(A_1 = A_2 \wedge p)(t \circ y)$ holds.
$\Rightarrow y \in (\sigma_{A_1=A_2}(e_2))(t)$, because $t \circ y$ satisfies $A_1 = A_2$.
$\Rightarrow \exists x \in (\sigma_{A_1=A_2}(e_2))(t) : p$, as y also satisfies p.

"⇐":

Let y be a tuple from $(\sigma_{A_1=A_2}(e_2))(t)$ for which p holds.
$\Rightarrow y \in e_2$
$\Rightarrow \exists x \in e_2 : (A_1 = A_2 \wedge p)(t \circ x)$, because y satisfies $t.A_1 = y.A_2$ and y satisfies p.

For lhs this means that $\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(t) = t$.
For rhs we get $t \ltimes_{A_1=A_2 \wedge p} e_2 = t =$ lhs.

Case 2: $/\exists x \in e_2 : (A_1 = A_2 \wedge p)(t \circ x)$ (which is equivalent to $/\exists x \in (\sigma_{A_1=A_2}(e_2))(t) : p$, as already shown above)

So for lhs we get $\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(t) = \epsilon$.
For rhs $t \ltimes_{A_1=A_2 \wedge p} e_2 = \epsilon =$ lhs.

A.2.4. Proof of Equivalence 4.4

$$\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(e_1)$$

Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:
lhs = rhs = ϵ

Inductive Hypothesis:
$\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(e_1)$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1 \oplus t) = \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(e_1 \oplus t) \\ \Leftrightarrow \quad & \sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) \oplus \sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(t) = \\ & \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(e_1) \oplus \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(t) \end{aligned}$$

As we know that $\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(e_1)$, we have to prove that $\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(t) = \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(t)$

Case 1: $aggr = \min, \theta \in \{>, \geq\}$

We have to look at a special case first, namely that there is no tuple in e_2 for which p holds. In that case we compare A_1 with an undefined value on the rhs. This can be solved in different ways, e.g. setting the result of the $\min$-operater on an empty sequence to NULL (and assuming that a comparison with a NULL value always returns false) or setting it to ∞.

Case 1(a): $\theta =$'$>$'

We show that $\exists x \in e_2 : (A_1 > A_2 \wedge p)(t \circ x) \Leftrightarrow t.A_1 > \min_{A_2}(\sigma_p(e_2))$

"$\Rightarrow$":

$\exists x \in e_2 : (A_1 > A_2 \wedge p)(t \circ x)$
$\Rightarrow t.A_1 > x.A_2$
Let $y \in e_2 : p \wedge y.A_2 = \min_{A_2}(e_2)$
$\Rightarrow x.A_2 \geq y.A_2$
$\Rightarrow t.A_1 > y.A_2$
$\Rightarrow t.A_1 > \min_{A_2}(e_2)$

"$\Leftarrow$":

$t.A_1 > \min_{A_2}(\sigma_p(e_2))$
Let $y = \min_{A_2}(\sigma_p(e_2))$
$\Rightarrow (A_1 > A_2 \wedge p)(t \circ y)$ is true
$\Rightarrow \exists x \in e_2 : (A_1 > A_2 \wedge p)(t \circ x)$

Case 1(b): $\theta =$'$\geq$'

Can be shown analogously to Case 1(b).

Case 2: $aggr = \max, \theta \in \{<, \leq\}$

Can be shown similarly to Case 1. However, the result of the $\max$-operater on an empty sequence has to be set to NULL or to $-\infty$.

A.2.5. Proof of Equivalence 4.5

$$\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1 = A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1 \theta A_2 \wedge p} e_2))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$.
Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:

lhs = rhs = ϵ

Inductive Hypothesis:

$\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1 = A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1 \theta A_2 \wedge p} e_2))$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& \sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1 \oplus t) = \\
& (e_1 \oplus t) \ltimes_{A_1 = A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1 \theta A_2 \wedge p} e_2)) \\
\Leftrightarrow \quad & \sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) \oplus \sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(t) = \\
& e_1 \ltimes_{A_1 = A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1 \theta A_2 \wedge p} e_2)) \oplus \\
& t \ltimes_{A_1 = A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1 \theta A_2 \wedge p} e_2))
\end{aligned}$$

For the rhs we now distinguish between two different cases:

Case 1: $\exists y \in e_1 : t.A_1 = y.A_1$

For $(e_1 \oplus t) \bowtie_{A_1 \theta A_2 \wedge p} e_2$ this means that either both y and t find a join partner in e_2 or none of them finds one. So we could just replace $(e_1 \oplus t)$ with e_1. This has no influence on the result of the semijoin with e_1.

Case 2: $\nexists y \in e_1 : t.A_1 = y.A_1$

If t does not find a join partner in e_2, it has no influence on the semijoin with e_1. If t finds a join partner in e_2, this has no influence on the semijoin either, as the value $t.A_1$ is not present in e_1. Again, we could just replace $(e_1 \oplus t)$ with e_1.

So the rhs is equal to:
$e_1 \ltimes_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1\theta A_2 \wedge p} e_2)) \oplus t \ltimes_{A_1=A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1\theta A_2 \wedge p} e_2))$
As we know that $\sigma_{\exists x \in (\sigma_{A_1\theta A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1\theta A_2 \wedge p} e_2))$ we have to prove that $\sigma_{\exists x \in (\sigma_{A_1\theta A_2}(e_2)):p}(t) = t \ltimes_{A_1=A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1\theta A_2 \wedge p} e_2)) = t \ltimes_{A_1=A_3} (\Pi_{A_3:A_1}((e_1 \bowtie_{A_1\theta A_2 \wedge p} e_2) \oplus (t \bowtie_{A_1\theta A_2 \wedge p} e_2)))$

Case 1: $\exists x \in e_2 : (A_1\theta A_2 \wedge p)(t \circ x)$

Obviously, $\exists x \in e_2 : (A_1\theta A_2 \wedge p)(t \circ x) \Leftrightarrow \exists x \in (\sigma_{A_1\theta A_2}(e_2))(t) : p$. (Proof is done analogously to the one in (4.3).)

For the lhs we get t.

For the rhs this means that t will find a join partner in e_2 and so t will find a join partner in the semijoin. So rhs = t.

Case 2: $\nexists x \in e_2 : (A_1\theta A_2 \wedge p)(t \circ x)$

For the lhs we get ϵ.

For the rhs this means that t will not find a join partner in e_2. Even if there is a tuple $y \in e_1$ that finds a join partner in e_2, $y.A_1$ has to have a different value than $t.A_1$ (otherwise, t would also have found a join partner in e_2). This tuple y will not be a join partner for t in the semijoin. So rhs = ϵ.

A.2.6. Proof of Equivalence 4.6

$$\Pi^D(e_1) \ltimes_{A_1=A_2} (\sigma_p(e_2)) \quad = \quad \sigma_{c>0}(E)$$

with $E = \Pi_{A_1:A_2}(\Gamma_{c;=A_2;count\circ\sigma_p}(e_2))$. The equivalence holds if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, and $\Pi^D(e_1) = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$.

Case 1: $e_2 = \epsilon$ ($\Rightarrow e_1 = \epsilon$)

lhs = rhs = ϵ

Case 2: $e_2 \neq \epsilon$ ($\Rightarrow e_1 \neq \epsilon$)

Let t_i be the i-th tuple in $\Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$ (again, we assume that Π^D is not order-preserving, but deterministic and idempotent). The order of the result of lhs is determined by the order of the tuples in $\Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$ (it is the concatenation of the results of processing t_1 to t_n). The result of processing the i-th tuple on lhs is

$$t_i \ltimes_{A_1=A_2} (\sigma_p(e_2)).$$

According to the definition of the semijoin operator:

$$\begin{aligned} &= t_i && \text{if } \exists x \in \sigma_p(e_2) : (A_1 = A_2)(t_i \circ x) \\ &= \epsilon && \text{if } \nexists x \in \sigma_p(e_2) : (A_1 = A_2)(t_i \circ x) \end{aligned}$$

Replacing the unary Γ with the binary one on rhs, we get

$$\begin{aligned} & \sigma_{c>0}(\Pi_{A_1:A_2}(\Gamma_{c;=A_2;count\circ\sigma_p}(e_2))) \\ = \; & \sigma_{c>0}(\Pi_{A_1:A_2}(\Pi_{A_2:A_2'}(\Pi^D_{A_2':A_2}(\Pi_{A_2}(e_2)) \\ & \Gamma_{c;A_2'=A_2;count\circ\sigma_p} e_2))) \\ = \; & \sigma_{c>0}(\Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2)) \\ & \Gamma_{c;A_1=A_2;count\circ\sigma_p} e_2). \end{aligned}$$

The order of the result of rhs is determined as in lhs, so the result of processing tuple t_i on rhs is

$$\begin{aligned} &= \sigma_{c>0}(t_i \circ \\ &\quad [c : count(\sigma_p(\sigma_{t_i|_{A_1}=A_2}(e_2)))]) \\ &= \sigma_{c>0}(t_i \circ \\ &\quad [c : count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i)]). \end{aligned}$$

According to the definition of σ this is

$$\begin{aligned} &= t_i \quad \text{if } count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i) > 0 \\ &= \epsilon \quad \text{if } count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i) = 0. \end{aligned}$$

We have to show that

$$\begin{aligned} &\exists x \in \sigma_p(e_2) : (A_1 = A_2)(t_i \circ x) \\ \Leftrightarrow \quad &count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i) > 0. \end{aligned}$$

"⇒":

$$\begin{aligned} &\exists x \in \sigma_p(e_2) : (A_1 = A_2)(t_i \circ x) \\ \Rightarrow \quad &x \in e_2 \end{aligned}$$

We know that x satisfies the predicate p, so

$$x \in \sigma_p(e_2).$$

Was also know that $t_i.A_1 = x.A_2$, so

$$x \in \sigma_{A_1=A_2}(\sigma_p(e_2))(t_i).$$

and, therefore, the $count$ is larger than 0.

"⇐":

$$\begin{aligned} &count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i) > 0 \\ \Rightarrow \quad &\exists x \in \sigma_{A_1=A_2}(\sigma_p(e_2)) : \text{true}(t_i) \\ \Rightarrow \quad &\exists x \in \sigma_p(e_2) : (A_1 = A_2)(t_i \circ x) \end{aligned}$$

A.2.7. Proof of Equivalence 4.13

$$\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{A_1=A_3} \Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1)))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$,
Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:
lhs = rhs = ϵ

Inductive Hypothesis:
$\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{A_1=A_3} \Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1)))$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$
\begin{aligned}
& \sigma_{\forall x \in (e_2):p}(e_1 \oplus t) = (e_1 \oplus t) \triangleright_{A_1=A_3} \Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1 \oplus t))) \\
\Leftrightarrow \quad & \sigma_{\forall x \in (e_2):p}(e_1) \oplus \sigma_{\forall x \in (e_2):p}(t) = \\
& e_1 \triangleright_{A_1=A_3} \Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1 \oplus t))) \oplus \\
& \quad t \triangleright_{A_1=A_3} \Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1 \oplus t))) \\
\Leftrightarrow \quad & \sigma_{\forall x \in (e_2):p}(e_1) \oplus \sigma_{\forall x \in (e_2):p}(t) = \\
& e_1 \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1))) \oplus (\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(t))))) \oplus \\
& t \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1))) \oplus (\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(t)))))
\end{aligned}
$$

Case 1: $\exists y \in e_1 : y.A_1 = t.A_1$

For the antijoin involving e_1 this means that $\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(t)))$ has no influence on the result (there is already a tuple with the same values in $\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1)))$ and duplicates have no influence on the antijoin).

Case 2: $\nexists y \in e_1 : y.A_1 = t.A_1$

In this case, $\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(t)))$ has no influence either, as the value for $t.A_1$ does not appear in e_1 and is irrelevant for the antijoin.

There are analogous arguments for the antijoin involving t, i.e. we can rewrite the above equivalence to:

$$
\begin{aligned}
& \sigma_{\forall x \in (e_2):p}(e_1) \oplus \sigma_{\forall x \in (e_2):p}(t) = \\
& e_1 \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1)))) \oplus t \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(t))))
\end{aligned}
$$

As we know that $\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1))))$, we have to prove that $\sigma_{\forall x \in (e_2):p}(t) = t \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(t))))$.

Case 1: $\forall x \in e_2(t) : p$

For the lhs, this means that t will pass the selection operator, so $\sigma_{\forall x \in (e_2):p}(t) = t$

On the rhs all tuples in $\Upsilon_{\mathcal{A}(e_2):e_2}(t)$ will be filtered out by $\sigma_{\neg p}$, which means that t will not find a join partner. As we have an antijoin, rhs = t.

Case 2: $\nexists x \in e_2(t) : p$

For the lhs, this means that $\sigma_{\exists x \in (e_2):p}(t) = \epsilon$.

For the rhs, this means that $\exists x \in e_2(t) : \neg p$. This tuple will pass the filter $\sigma_{\neg p}$, which means that t will find a join partner. So the result of the antijoin is empty: rhs = ϵ.

A.2.8. Proof of Equivalence 4.14

$$\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{\neg p} e_2$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$.
Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:

lhs = rhs = ϵ

Inductive Hypothesis:

$\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{\neg p} e_2$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\forall x \in (e_2):p}(e_1 \oplus t) = (e_1 \oplus t) \triangleright_{\neg p} e_2 \\ \Leftrightarrow \quad & \sigma_{\forall x \in (e_2):p}(e_1) \oplus \sigma_{\forall x \in (e_2):p}(t) = \\ & e_1 \triangleright_{\neg p} e_2 \oplus t \triangleright_{\neg p} e_2 \end{aligned}$$

As we know that $\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{\neg p} e_2$, we have to prove that $\sigma_{\forall x \in (e_2):p}(t) = t \triangleright_{\neg p} e_2$

Case 1: $\exists x \in e_2 : (\neg p)(t \circ x)$

For the lhs, this means that t will not pass the selection operator, so $\sigma_{\exists x \in (e_2):p}(t) = \epsilon$

For the rhs, this means that t finds a join partner in e_2 and consequently will be filtered out by the antijoin (definition of antijoin), so rhs = ϵ

Case 2: $\nexists x \in e_2 : (\neg p)(t \circ x)$

For the lhs, this means that $\sigma_{\exists x \in (e_2):p}(t) = t$.

For the rhs, t will not find a join partner and, therefore, will stay (due to the antijoin), so rhs = t.

A.2.9. Proof of Equivalence 4.15

$$\sigma_{\forall x \in (\sigma_{A_1 = A_2}(e_2)):p}(e_1) = e_1 \triangleright_{A_1 = A_2 \wedge \neg p} e_2$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$.
Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:

lhs = rhs = ϵ

Inductive Hypothesis:

$\sigma_{\forall x \in (\sigma_{A_1 = A_2}(e_2)):p}(e_1) = e_1 \triangleright_{A_1 = A_2 \wedge \neg p} e_2$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\forall x \in (\sigma_{A_1 = A_2}(e_2)):p}(e_1 \oplus t) = \\ & (e_1 \oplus t) \triangleright_{A_1 = A_2 \wedge \neg p} e_2 \\ \Leftrightarrow \quad & \sigma_{\forall x \in (\sigma_{A_1 = A_2}(e_2)):p}(e_1) \oplus \sigma_{\forall x \in (\sigma_{A_1 = A_2}(e_2)):p}(t) = \\ & e_1 \triangleright_{A_1 = A_2 \wedge \neg p} e_2 \oplus t \triangleright_{A_1 = A_2 \wedge \neg p} e_2 \end{aligned}$$

As we know that $\sigma_{\forall x \in (\sigma_{A_1 = A_2}(e_2)):p}(e_1) = e_1 \triangleright_{A_1 = A_2 \wedge \neg p} e_2$, we have to prove that $\sigma_{\forall x \in (\sigma_{A_1 = A_2}(e_2)):p}(t) = t \triangleright_{A_1 = A_2 \wedge \neg p} e_2$.

Case 1: $\nexists x \in e_2 : (A_1 = A_2 \wedge \neg p)(t \circ x)$

First of all, we show that $/\exists x \in e_2 : (A_1 = A_2 \wedge \neg p)(t \circ x) \Leftrightarrow \forall x \in (\sigma_{A_1 = A_2}(e_2))(t) : p$.

Case 1(a): $e_2 = \epsilon$

lhs = rhs = true

Case 1(b): $e_2 \neq \epsilon$

"⇒":

Let y be an arbitrary tuple from $Z = \{z | z \in e_2 \wedge z.A_2 \neq t.A_1\}$.
$\Rightarrow y \notin (\sigma_{A_1 = A_2}(e_2))(t)$
$\Rightarrow$ Such a tuple y cannot be the cause for $\forall x \in (\sigma_{A_1 = A_2}(e_2))(t) : p =$

false.

So, let y' be an arbitrary tuple from $Z' = e_2 \setminus Z$ (i.e. $Z' = \{z | z \in e_2 \wedge z.A_2 = t.A_1\}$).
$\Rightarrow y'$ satisfies p, because there is no tuple in e_2 for which $(A_1 = A_2)$ **and** $\neg p$ holds.
$\Rightarrow$ No tuple y' can be the cause for $\forall x \in (\sigma_{A_1=A_2}(e_2))(t) : p =$ false.

As $Z \cup Z' = e_2$, there can be no tuple in e_2 which causes $\forall x \in (\sigma_{A_1=A_2}(e_2))(t) : p$ to be false.
$\Rightarrow \forall x \in (\sigma_{A_1=A_2}(e_2))(t) : p$ holds.

"$\Leftarrow$":

Let us assume that $\exists x \in e_2 : (A_1 = A_2 \wedge \neg p)(t \circ x)$.
$\Rightarrow x \in (\sigma_{A_1=A_2}(e_2))(t)$
As x satisfies $\neg p$, it cannot satisfy p.
$\Rightarrow \not\forall x \in (\sigma_{A_1=A_2}(e_2))(t) : p$, which contradicts our prerequisite.
Therefore, $\nexists x \in e_2 : (A_1 = A_2 \wedge \neg p)(t \circ x)$.

For lhs this means that $\sigma_{\forall x \in (\sigma_{A_1=A_2}(e_2)):p}(t) = t$.
For rhs we get $t \triangleright_{A_1=A_2 \wedge \neg p} e_2 = t =$ lhs.

Case 2: $\exists x \in e_2 : (A_1 = A_2 \wedge \neg p)(t \circ x)$ (which is equivalent to $\not\forall x \in (\sigma_{A_1=A_2}(e_2))(t) : p$. as already shown above)
So for lhs we get $\sigma_{\forall x \in (\sigma_{A_1=A_2}(e_2)):p}(t) = \epsilon$.
For rhs $t \triangleright_{A_1=A_2 \wedge \neg p} e_2 = \epsilon =$ lhs.

A.2.10. Proof of Equivalence 4.16

$$\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \neg\theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(e_1)$$

Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:
lhs = rhs = ϵ

Inductive Hypothesis:
$\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2))p}(e_1) = \sigma_{A_1 \neg\theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(e_1)$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1 \oplus t) = \sigma_{A_1 \neg\theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(e_1 \oplus t) \\ \Leftrightarrow \quad & \sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) \oplus \sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(t) = \\ & \sigma_{A_1 \neg\theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(e_1) \oplus \sigma_{A_1 \neg\theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(t) \end{aligned}$$

As we know that $\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \neg\theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(e_1)$, we have to prove that $\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(t) = \sigma_{A_1 \neg\theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(t)$

Case 1: $aggr = \min, \theta \in \{>, \geq\}$
We have to look at a special case first, namely that there is no tuple in e_2 for which $\neg p$ holds. In that case we compare A_1 with an undefined value on the rhs. This can be solved in different ways, e.g. always returning **true** when comparing A_1 with an undefined value or setting $\min$ to $-\infty$. Note that this is different to Equivalence 4.4.

Case 1(a): $\theta =$'$>$'

We show that $\forall x \in (\sigma_{A_1 > A_2}(e_2))(t) : p \Leftrightarrow t.A_1 \leq \min_{A_2}(\sigma_{\neg p}(e_2))$

"$\Rightarrow$":

Assume that $t.A_1 > \min_{A_2}(\sigma_{\neg p}(e_2))$
$\Rightarrow \exists x \in e_2 : (A_1 > A_2 \wedge \neg p)(t \circ x)$
$\Rightarrow x \in (\sigma_{A_1 > A_2}(e_2))(t)$
As x satisfies $\neg p$, it cannot satisfy p
$\Rightarrow \nexists x \in (\sigma_{A_1 > A_2}(e_2))(t) : p$, which is a contradiction
$\Rightarrow t.A_1 \leq \min_{A_2}(e_2)$

"$\Leftarrow$":

$t.A_1 \leq \min_{A_2}(\sigma_p(e_2))$
$\Rightarrow \nexists x \in (\sigma_{A_1 > A_2}(e_2))(t) : p$
$\Rightarrow \forall x \in (\sigma_{A_1 > A_2}(e_2))(t) : p$

Case 1(b): $\theta =$'$\geq$'

Can be shown analogously to Case 1(b).

Case 2: $aggr = \max, \theta \in \{<, \leq\}$

Can be shown similarly to Case 1. However, the result of the $\max$-operater on an empty sequence has to be set to ∞ (or the comparison with A_1 always returns true).

A.2.11. Proof of Equivalence 4.17

$$\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = e_1 \triangleright_{A_1 = A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1 \theta A_2 \wedge \neg p} e_2))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$.
Proof by Induction: over the length of the sequence e_1

Base Case: $e_1 = \epsilon$:

lhs = rhs = ϵ

Inductive Hypothesis:

$\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = e_1 \triangleright_{A_1 = A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1 \theta A_2 \wedge \neg p} e_2))$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1 \oplus t) = \\ & (e_1 \oplus t) \triangleright_{A_1 = A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1 \theta A_2 \wedge \neg p} e_2)) \\ \Leftrightarrow \quad & \sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) \oplus \sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(t) = \\ & e_1 \triangleright_{A_1 = A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1 \theta A_2 \wedge \neg p} e_2)) \oplus \\ & t \triangleright_{A_1 = A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1 \theta A_2 \wedge \neg p} e_2)) \end{aligned}$$

For the rhs we now distinguish between two different cases:

Case 1: $\exists y \in e_1 : t.A_1 = y.A_1$

For $(e_1 \oplus t) \bowtie_{A_1 \theta A_2 \wedge \neg p} e_2$ this means that either both y and t find a join partner in e_2 or none of them finds one. So we can replace $(e_1 \oplus t)$ with e_1. This has no influence on the result of the antijoin with e_1.

Case 2: $\nexists y \in e_1 : t.A_1 = y.A_1$

If t does not find a join partner in e_2, it has no influence on the antijoin with e_1 (there is no tuple in e_1 to join with anyway). If t finds a join partner in e_2, this also has no influence on the antijoin, as the value $t.A_1$ is not present in e_1. Again, we can replace $(e_1 \oplus t)$ with e_1.

So the rhs is equal to:
$e_1 \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1\theta A_2 \wedge \neg p} e_2)) \oplus$
$t \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1\theta A_2 \wedge \neg p} e_2))$

As we know that
$\sigma_{\forall x \in (\sigma_{A_1\theta A_2}(e_2)):p}(e_1) = e_1 \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1\theta A_2 \wedge \neg p} e_2))$,
we have to prove that
$\sigma_{\forall x \in (\sigma_{A_1\theta A_2}(e_2)):p}(t) = e_1 \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1\theta A_2 \wedge \neg p} e_2)) \oplus$
$t \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}((e_1 \oplus t) \bowtie_{A_1\theta A_2 \wedge \neg p} e_2))$.

Case 1: $\nexists x \in e_2 : (A_1\theta A_2 \wedge \neg p)(t \circ x)$

Obviously, $\nexists x \in e_2 : (A_1\theta A_2 \wedge \neg p)(t \circ x) \Leftrightarrow \forall x \in (\sigma_{A_1\theta A_2}(e_2))(t) : p$. (Proof is done analogously to the one in (4.15).)

For the lhs we get t.

For the rhs this means that t will not find a join partner in e_2. Even if there is a tuple $y \in e_1$ that finds a join partner in e_2, $y.A_1$ has to have a different value than $t.A_1$ (otherwise, t would also have found a join partner in e_2). This tuple y will not be a join partner for t in the antijoin. So rhs = t.

Case 2: $\exists x \in e_2 : (A_1\theta A_2 \wedge \neg p)(t \circ x)$

For the lhs we get ϵ.

For the rhs this means that t will find a join partner in e_2, which will lead to an unsatisfied predicate for the antijoin. So rhs = ϵ.

A.2.12. Proof of Equivalence 4.18

$$\Pi^D(e_1) \triangleright_{A_1=A_2} (\sigma_p(e_2)) \quad = \quad \sigma_{c=0}(E)$$

with $E = \Pi_{A_1:A_2}(\Gamma_{c;=A_2;count\circ\sigma_p}(e_2))$. The equivalence holds if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, and $\Pi^D(e_1) = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$.

Case 1: $e_2 = \epsilon$ ($\Rightarrow e_1 = \epsilon$)

lhs = rhs = ϵ

Case 2: $e_2 \neq \epsilon$ ($\Rightarrow e_1 \neq \epsilon$)

Let t_i be the i-th tuple in $\Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$ (with a non-order-preserving, deterministic, idempotent Π^D). The order of the result of lhs is determined by the order of the tuples in $\Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$. Processing the i-th tuple on lhs results in

$$t_i \triangleright_{A_1=A_2} (\sigma_p(e_2)).$$

In accordance to the definition of the semijoin operator:

$$\begin{aligned} &= t_i && \text{if } \nexists x \in \sigma_p(e_2) : (A_1 = A_2)(t_i \circ x) \\ &= \epsilon && \text{if } \exists x \in \sigma_p(e_2) : (A_1 = A_2)(t_i \circ x) \end{aligned}$$

Replacing the unary Γ with the binary one on rhs, we get

$$\begin{aligned} & \sigma_{c=0}(\Pi_{A_1:A_2}(\Gamma_{c;=A_2;count\circ\sigma_p} e_2)) \\ = \; & \sigma_{c=0}(\Pi_{A_1:A_2}(\Pi_{A_2:A'_2}(\Pi^D_{A'_2:A_2}(\Pi_{A_2}(e_2))) \\ & \Gamma_{c;A'_2=A_2;count\circ\sigma_p}(e_2))) \\ = \; & \sigma_{c=0}(\Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2)) \\ & \Gamma_{c;A_1=A_2;count\circ\sigma_p}(e_2)). \end{aligned}$$

The order of the result of rhs is determined as in lhs, so the result of processing tuple t_i on rhs is

$$\begin{aligned} &= \sigma_{c=0}(t_i \circ \\ &\quad [c : count(\sigma_p(\sigma_{t_i|_{A_1}=A_2}(e_2)))]) \\ &= \sigma_{c=0}(t_i \circ \\ &\quad [c : count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i)]). \end{aligned}$$

According to the definition of σ, this is

$$\begin{aligned} &= t_i \quad \text{if } count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i) = 0 \\ &= \epsilon \quad \text{if } count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i) > 0. \end{aligned}$$

We have to show that

$$\begin{aligned} &\nexists x \in \sigma_p(e_2) : (A_1 = A_2)(t_i \circ x) \\ \Leftrightarrow \quad &count(\sigma_{A_1=A_2}(\sigma_p(e_2)))(t_i) = 0, \end{aligned}$$

which has already been done for the previous equivalence.

A.2.13. Proof of Equivalence 4.23

$$\chi_{g:f(\sigma_p(e_2))}(e_1) \quad = \quad e_1 \Gamma_{g;\mathcal{A}(e_1)=A_1';f}(\Pi_{A_1':\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))))$$

if $g \notin \mathcal{A}(e_1), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$.

Case 1: $e_1 = \epsilon$
From the definition of χ and binary Γ immediately follows: lhs = ϵ and rhs = ϵ.

Case 2: $e_1 \neq \epsilon$
Let t_i be the i-th tuple of e_1. As the χ operator traverses e_1 tuple by tuple (while preserving the order), the i-th tuple of lhs is equal to

$$t_i \circ [g : f(\sigma_p(e_2))(t_i)].$$

The binary Γ operator also traverses e_1 tuplewise, so the i-th tuple of rhs is equal to

$$\begin{aligned} &t_i \circ [g : f(\sigma_{t_i|_{\mathcal{A}(e_1)}=A_1'}(\Pi_{A_1':\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1))))))] \\ = \quad &t_i \circ [g : f(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\sigma_{t_i|_{\mathcal{A}(e_1)}=A_1'}(\Pi_{A_1':\mathcal{A}(e_1)}(\Pi^D_{\mathcal{A}(e_1)}(e_1))))))] \\ = \quad &t_i \circ [g : f(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(t_i)))] \end{aligned}$$

As $(e_2)(t_i)$ contains the same tuples from e_2 as $\Upsilon_{\mathcal{A}(e_2):e_2}(t_i)$, lhs is equal to rhs.

A.2.14. Proof of Equivalence 4.24

$$\begin{aligned} \chi_{g:f(\sigma_p(e_2))}(e_1) \quad = \quad &\Pi_{\overline{A_3}}(e_1 \, \bowtie^{g:f(\epsilon)}_{\mathcal{A}(e_1)=A_3} (\Pi_{A_3:\mathcal{A}(e_1)}(\Gamma_{g;=\mathcal{A}(e_1);f}(\\ &\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1))))))) \end{aligned}$$

if $g \notin \mathcal{A}(e_1), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$.

Case 1: $e_1 = \epsilon$
from the definition of χ, Π and $\bowtie$ immediately follows: lhs = ϵ and rhs = ϵ.

Case 2: $e_1 \neq \epsilon$

Let t_i be the i-th tuple in e_1; e_1 imposes its order upon lhs and rhs, as χ and $\bowtie$ are order-preserving (for the outer join the expression e_1 left of the $\bowtie$ is relevant). Transforming the unary Γ into a binary Γ, we get:

$$\Pi_{\overline{A_3}}(e_1 \; \bowtie^{g:f(\epsilon)}_{\mathcal{A}(e_1)=A_3} \; (\Pi_{A_3:\mathcal{A}(e_1)}(\Pi_{\mathcal{A}(e_1):A'_1}(h(e_1,e_2)))))$$

with $h(e_1,e_2)$ equal to

$$\overbrace{\Pi^D_{A'_1:\mathcal{A}(e_1)}(\Pi_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))))}^{h_2(e_1,e_2)} \qquad \Gamma_{g;A'_1=\mathcal{A}(e_1);f}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1))))$$

$h_2(e_1,e_2)$ being the first operand of the binary Γ

Case 2(a): $\nexists x \in \sigma_p(e_2(t_i))$:

For the tuple t_i this means that no tuples are produced in the (dependent) expression e_2 that satisfy the predicate p.
For lhs we have

$$\begin{aligned} & t_i \circ [g : f(\sigma_p(e_2))(t_i)] \\ = \; & t_i \circ [g : f(\epsilon)]. \end{aligned}$$

For the right hand side (rhs) we get

$$t_i \circ [g : f(\epsilon)]$$

because $\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(t_i)))$ does not produce a tuple. Consequently, $h_2(e_1,e_2)$ in the binary Γ the group for $t_i.\mathcal{A}(e_1)$ is empty, resulting in $h(e_1,e_2) = \epsilon$ for these attribute values. So in the outer join t_i does not find a join partner, resulting in $t_i \circ [g : f(\epsilon)]$.

Case 2(b): $\exists x \in \sigma(e_2(t_i))$:

For the lhs we get

$$t_i \circ [g : f(\sigma_p(e_2))(t_i)]$$

For the rhs this means that we are looking for the join partner of t_i in the outer join expression. Let t_j be the tuple in $h_2(e_1,e_2)$ with $t_j.A'_1 = t_i.\mathcal{A}(e_1)$. As $\exists x \in \sigma(e_2(t_i))$, in the binary grouping operator there will be one group for the attribute values of $t_i.\mathcal{A}(e_1)$, namely:

$$\begin{aligned} & t_j \circ [g : f(\sigma_{t_j|A'_1=\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))))] \\ = \; & t_j \circ [g : f(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\sigma_{t_j|A'_1=\mathcal{A}(e_1)}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))))] \end{aligned}$$

As $t_j.A'_1 = t_i.\mathcal{A}(e_1)$, this is equal to

$$t_j \circ [g : f(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(t_i)))]$$

Also, this is the only join partner for t_i in the outer join expression (all other tuples in the result of $h(e_1,e_2)$ have other values for A'_1). After joining this to t_i, renaming, and projecting away unnecessary attributes we get:

$$t_i \circ [g : f(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(t_i)))]$$

As $(e_2)(t_i)$ contains the same tuples from e_2 as $\Upsilon_{x:e2}(t_i)$, lhs is equal to rhs.

A.2.15. Proof of Equivalence 4.25

$$\chi_{g:f(\sigma_p(e_2))}(e_1) \quad = \quad e_1\Gamma_{g;\mathcal{A}(e_1)=A_1';f}(\Pi_{A_1':\mathcal{A}(e_1)}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))$$

if $g \notin \mathcal{A}(e_1)$, $\mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset$.

Case 1: $e_1=\epsilon$
From the definition of χ and binary Γ immediately follows: lhs = ϵ and rhs = ϵ.

Case 2: $e_1\neq\epsilon$
Let t_i be the i-th tuple of e_1. As the χ operator traverses e_1 tuple by tuple (while preserving the order), the i-th tuple of lhs is equal to

$$t_i \circ [g : f(\sigma_p(e_2))(t_i)].$$

The binary Γ operator also traverses e_1 tuplewise, so the i-th tuple of rhs is equal to

$$\begin{aligned} & t_i \circ [g : f(\sigma_{t_i|_{\mathcal{A}(e_1)}=A_1'}(\Pi_{A_1':\mathcal{A}(e_1)}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2))))] \\ = \quad & t_i \circ [g : f(\sigma_p(\sigma_{t_i|_{\mathcal{A}(e_1)}=A_1'}(\Pi_{A_1':\mathcal{A}(e_1)}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))\times e_2))] \\ = \quad & t_i \circ [g : f(\sigma_p(t_i\times e_2))] \end{aligned}$$

As $(e_2)(t_i)$ contains the same tuples from e_2 as $t_i \times e_2$ (e_2 can be evaluated independently of e_1), lhs is equal to rhs.

A.2.16. Proof of Equivalence 4.26

$$\begin{aligned} \chi_{g:f(\sigma_p(e_2))}(e_1) \quad = \quad & \Pi_{\overline{A_3}}(e_1 \ltimes^{g:f(\epsilon)}_{\mathcal{A}(e_1)=A_3} (\Pi_{A_3:\mathcal{A}(e_1)}(\\ & \Gamma_{g;=\mathcal{A}(e_1);f}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2))))) \end{aligned}$$

if $g \notin \mathcal{A}(e_1)$, $\mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset$.

Case 1: $e_1=\epsilon$
From the definition of χ, Π and $\ltimes$ immediately follows: lhs = ϵ and rhs = ϵ.

Case 2: $e_1\neq\epsilon$
Let t_i be the i-th tuple in e_1; e_1 imposes its order upon lhs and rhs, as χ and $\ltimes$ are order-preserving (for the outer join the expression e_1 left of the $\ltimes$ is relevant). Transforming the unary Γ into a binary Γ, we get:

$$\Pi_{\overline{A_3}}(e_1 \ltimes^{g:f(\epsilon)}_{\mathcal{A}(e_1)=A_3} (\Pi_{A_3:\mathcal{A}(e_1)}(\Pi_{\mathcal{A}(e_1):A_1'}(h(e_1,e_2)))))$$

with $h(e_1,e_2)$ equal to

$$\overbrace{\Pi^D_{A_1':\mathcal{A}(e_1)}(\Pi_{\mathcal{A}(e_1)}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))}^{h_2(e_1,e_2)}\Gamma_{g;A_1'=\mathcal{A}(e_1);f}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2))$$

$h_2(e_1,e_2)$ being the first operand of the binary Γ

Case 2(a): $\nexists x \in \sigma_p(e_2(t_i))$:
For the tuple t_i this means that no tuples are produced in the (independent) expression e_2 that satisfy the predicate p.
For lhs we have

$$\begin{aligned} & t_i \circ [g : f(\sigma_p(e_2))(t_i)] \\ = \quad & t_i \circ [g : f(\epsilon)]. \end{aligned}$$

For the right hand side (rhs) we get

$$t_i \circ [g : f(\epsilon)]$$

because $\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(t_i) \times e_2)$ does not produce a tuple. Consequently, $h_2(e_1, e_2)$ in the binary Γ the group for $t_i.\mathcal{A}(e_1)$ is empty, resulting in $h(e_1, e_2) = \epsilon$ for these attribute values. So in the outer join t_i does not find a join partner, resulting in $t_i \circ [g : f(\epsilon)]$.

Case 2(b): $\exists x \in \sigma(e_2(t_i))$:
For the lhs we get

$$t_i \circ [g : f(\sigma_p(e_2))(t_i)]$$

For the rhs this means that we are looking for the join partner of t_i in the outer join expression. Let t_j be the tuple in $h_2(e_1, e_2)$ with $t_j.A'_1 = t_i.\mathcal{A}(e_1)$. As $\exists x \in \sigma(e_2(t_i))$, in the binary grouping operator there will be one group for the attribute values of $t_i.\mathcal{A}(e_1)$, namely:

$$\begin{aligned} & t_j \circ [g : f(\sigma_{t_j|A'_1=\mathcal{A}(e_1)}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1) \times e_2)))] \\ = \quad & t_j \circ [g : f(\sigma_p(\sigma_{t_j|A'_1=\mathcal{A}(e_1)}(\Pi^D_{\mathcal{A}(e_1)}(e_1) \times e_2)))] \end{aligned}$$

As $t_j.A'_1 = t_i.\mathcal{A}(e_1)$, this is equal to

$$t_j \circ [g : f(\sigma_p(t_i \times e_2))]$$

Also, this is the only join partner for t_i in the outer join expression (all other tuples in the result of $h(e_1, e_2)$ have other values for A'_1). After joining this to t_i, renaming, and projecting away unnecessary attributes we get:

$$t_i \circ [g : f(\sigma_p(t_i) \times e_2)]$$

As $(e_2)(t_i)$ contains the same tuples from e_2 as $t_i \times e_2$, lhs is equal to rhs.

A.2.17. Proof of Equivalence 4.27

$$\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) = e_1 \Gamma_{g;A_1\theta A_2;f} e_2$$

if $A_i \subseteq \mathcal{A}(e_i)$, $g \notin A_1 \cup A_2$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, and $A_1 \cap A_2 = \emptyset$.

Case 1: $e_1 = \epsilon$
From the definition of χ and binary Γ immediately follows: lhs = ϵ and rhs = ϵ.

Case 2: $e_1 \neq \epsilon$
Let t_i be the i-th tuple of e_1, $t_i = \alpha(\underbrace{\tau(\tau(\ldots\tau}_{i-1}(e_1)\ldots)))$.
As the χ operator traverses e_1 tuple by tuple, the i-th tuple of lhs is equal to

$$t_i \circ [g : f(\sigma_{A_1\theta A_2}(e_2))(t_i)].$$

The binary Γ operator also traverses e_1 tuplewise, so the i-th tuple of rhs is equal to

$$\begin{aligned} & t_i \circ [g : f(\sigma_{t_i|_{A_1}\theta A_2}(e_2))] \\ = \quad & t_i \circ [g : f(\sigma_{A_1\theta A_2}(e_2))(t_i)] \end{aligned}$$

as $A_1 \subseteq \mathcal{A}(e_1)$.

A.2.18. Proof of Equivalence 4.28

$$\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) = \Pi_{\overline{A_3}}(e_1 \ \mathbb{M}^{g:f(\epsilon)}_{A_1=A_3} \ (\Pi_{A_3:A_1}(\Gamma_{g;=A_1;f}(\Pi^D_{A_1}(e_1) \bowtie_{A_1\theta A_2} e_2))))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, $A_1 \cap A_2 = \emptyset$, and $g \notin A_1 \cup A_2$.

Case 1: $e_1 = \epsilon$

from the definition of χ, Π and $\mathbb{M}$ immediately follows: lhs = ϵ and rhs = ϵ.

Case 2: $e_1 \neq \epsilon$

Let t_i be the i-th tuple in e_1; e_1 imposes its order upon lhs and rhs, as χ and $\mathbb{M}$ are order-preserving. Transforming the unary Γ into a binary Γ, we get:

$$\Pi_{\overline{A_3}}(e_1 \ \mathbb{M}^{g:f(\epsilon)}_{A_1=A_3} \ (\Pi_{A_3:A_1}(\Pi_{A_1:A'_1}(h(e_1,e_2)))))$$

with $h(e_1, e_2)$ equal to

$$\overbrace{\Pi^D_{A'_1:A_1}(\Pi_{A_1}(\Pi^D_{A_1}(e_1) \bowtie_{A_1\theta A_2} e_2))}^{h_2(e_1,e_2)} \Gamma_{g;A'_1=A_1;f}(\Pi^D_{A_1}(e_1) \bowtie_{A_1\theta A_2} e_2)$$

$h_2(e_1, e_2)$ being the first operand of the binary Γ

Case 2(a): $\nexists x \in e_2 : t_i.A_1\theta x.A_2$ holds

($\Rightarrow t_i \bowtie_{A_1=A_3} \Pi_{A_3:A_1}(\Pi_{A_1:A'_1}(h(e_1,e_2))) = \epsilon$, as both operands of Γ are empty)

For lhs we have

$$\begin{aligned} & t_i \circ [g : f(\sigma_{A_1\theta A_2}(e_2))(t_i)] \\ = \ & t_i \circ [g : f(\epsilon)]. \end{aligned}$$

For the right hand side (rhs) we get

$$t_i \circ [g : f(\epsilon)]$$

because the join in $h_2(e_1, e_2)$ does not produce a tuple with a value of $t_i.A_1$ for attribute A_1. So the group for $t_i.A_1$ is empty and in the outer join t_i is joined with an empty tuple.

Case 2(b): $\exists x \in e_2 : t_i.A_1\theta x.A_2$ holds

For the left hand side (lhs) we have

$$t_i \circ [g : f(\sigma_{A_1\theta A_2}(e_2))(t_i)]$$

For the right hand side (rhs), we now have to show that t_i finds a join partner in $h(e_1, e_2)$ and that it is equal to $[f(\sigma_{A_1\theta A_2}(e_2))(t_i)]$.

Let t_j be the tuple in $\Pi^D_{A_1}(e_1)$ with $t_j.A_1 = t_i.A_1$. As we know that $\exists x \in e_2$ for which $t_i.A_1\theta x.A_2$ holds, t_j finds at least one join partner in e_2. We join t_j with all corresponding tuples and then project with duplicate elimination to A_1, so $h_2(e_1, e_2)$ contains one tuple t'_j with $t'_j.A_1 = t_i.A_1$ (which is relevant for the join with t_i). Let us now look at the group generated by the grouping operator for t'_j. The tuple for the value $t_i.A_1$ in $h(e_1, e_2)$ is equal to

$$\begin{aligned} & t'_j \circ [g : f(\sigma_{t'_j|A'_1=A_1}(\Pi^D_{A_1}(e_1) \bowtie_{A_1\theta A_2} e_2))] \\ = \ & t'_j \circ [g : f(\sigma_{A'_1=A_1}(\Pi^D_{A_1}(e_1) \bowtie_{A_1\theta A_2} e_2))(t'_j)] \end{aligned}$$

Let t_k be the tuple in $\Pi^D_{A_1}(e_1)$ for which $t_k.A_1 = t'_j.A'_1 = t_i.A_1$ (all other tuples will be filtered out by $\sigma_{A'_1=A_1}$). t_k will be joined to the following tuples in e_2:

$$t_k \bowtie_{A_1 \theta A_2} e_2$$

For the concatenation with t'_j this means

$$t'_j \circ [g : f(\sigma_{A'_1=A_1}(t_k \bowtie_{A_1 \theta A_2} (e_2)))(t'_j)]$$

As f only references attributes of e_2 and A_1 (or A'_1, respectively), $t'_j.A'_1 = t_k.A_1$, and the join is order-preserving, this is equal to

$$t'_j \circ [g : f(\sigma_{A'_1 \theta A_2}(e_2))(t'_j)]$$

Finally, after renaming ($\Pi_{A_3:A_1}$ and $\Pi_{A_1:A'_1}$) this is joined to t_i, and unnecessary attributes are eliminated by projection ($\Pi_{\overline{A_3}}$). So we get:

$$t_i \circ [g : f(\sigma_{A_1 \theta A_2}(e_2))(t_i)]$$

A.2.19. Proof of Equivalence 4.29

$$\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) = \Pi_{\overline{A_2}}(e_1 \bowtie^{g:f(\epsilon)}_{A_1=A_2} (\Gamma_{g;=A_2;f}(e_2)))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, $A_1 \cap A_2 = \emptyset$, and $g \notin A_1 \cup A_2$.

Case 1: $e_1 = \epsilon$

from the definition of χ, Π and $\bowtie$ immediately follows: lhs = ϵ and rhs = ϵ.

Case 2: $e_1 \neq \epsilon$

Let t_i be the i-th tuple in e_1 and

$$\begin{aligned} h(e_2) &= \Gamma_{g;=A_2;f}(e_2) \\ &= \Pi_{A_2:A'_2}(\Pi^D_{A'_2:A_2}(\Pi_{A_2}(e_2))\Gamma_{g;A'_2=A_2;f}e_2). \end{aligned}$$

e_2 is projected on A_2 with a duplicate elimination, so each value of A_2 appears only once in $h(e_2)$. Let t'_j be the j-th tuple in $\Pi^D_{A'_2:A_2}(\Pi_{A_2}(e_2))$. The j-th tuple in $h(e_2)$ then is

$$\begin{aligned} & \Pi_{A_2:A'_2}(t'_j \circ [g : f(\sigma_{t_j|_{A'_2}=A_2}(e_2))]) \\ = \; & \Pi_{A_2:A'_2}(t'_j \circ [g : f(\sigma_{A'_2=A_2}(e_2))(t'_j)]). \end{aligned}$$

Each tuple t_i in e_1 joins with at most one tuple in $h(e_2)$ with join predicate $A_1 = A_2$. If no join partner is found in $h(e_2)$, then an empty tuple is concatenated to t_i via the outer join operator. For each tuple t_i in e_1 we have the corresponding tuple at the i-th position after the outer join.

Case 2(a): $\nexists x \in e_2 : t_i.A_1 = x.A_2$

($\Rightarrow t_i \bowtie_{A_1=A_2} h(e_2) = \epsilon$)

For lhs we have

$$\begin{aligned} & t_i \circ [g : f(\sigma_{A_1=A_2}(e_2))(t_i)] \\ = \; & t_i \circ [g : f(\epsilon)]. \end{aligned}$$

For the right hand side (rhs) we get

$$\begin{aligned} & \Pi_{\overline{A_2}}(t_i \circ \perp_{A_2} \circ [g : f(\epsilon)]) \\ = \; & t_i \circ [g : f(\epsilon)]. \end{aligned}$$

Case 2(b): $\exists x \in e_2 : t_i.A_1 = x.A_2$
($\Rightarrow t_i \bowtie_{A_1=A_2} h(e_2) \neq \epsilon$)
For the left hand side (lhs) we have

$$t_i \circ [g : f(\sigma_{A_1=A_2}(e_2))(t_i)].$$

We now turn to rhs. Let t''_k be the tuple from $h(e_2)$ for which $t''_k.A_2 = t_i.A_1$ (all other tuples in $h(e_2)$ are irrelevant for the join). Therefore, rhs is equal to

$$\begin{aligned} & \Pi_{\overline{A_2}}(t_i \bowtie_{A_1=A_2} h(e_2)) \\ = \; & \Pi_{\overline{A_2}}(t_i \circ t''_k) \\ = \; & \Pi_{\overline{A_2}}(t_i \circ \Pi_{A_2:A'_2}(t'_k \circ [g : f(\sigma_{A'_2=A_2}(e_2))(t'_k)])). \end{aligned}$$

As $t_i.A_1 = t''_k.A_2 = t'_k.A'_2$ and we project away A'_2 (after renaming it to A_2), we get

$$t_i \circ [g : f(\sigma_{A_1=A_2}(e_2))(t_i)].$$

A.2.20. Proof of Equivalence 4.30

$$\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) = \Pi_{A_1:A_2}(\Gamma_{g;=A_2;f}(e_2))$$

if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, $A_1 \cap A_2 = \emptyset$, $g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$, and $e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$ (this implies that $A_1 = \mathcal{A}(e_1)$)

Case 1: $e_2 = \epsilon$ ($\Rightarrow e_1 = \epsilon$)
From the definition of χ and unary Γ immediately follows: lhs = ϵ and rhs = ϵ.

Case 2: $e_2 \neq \epsilon$ ($\Rightarrow e_1 \neq \epsilon$)
The Π^D in $\Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$ does not necessarily preserve the original order in e_2, but we assume that it is a deterministic operator, i.e., for the same input we always get the same output order. Let t_i be the i-th tuple in $\Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$.
So, the i-th tuple in lhs is

$$t_i \circ [g : f(\sigma_{A_1=A_2}(e_2))(t_i)].$$

Replacing the unary Γ in rhs with the binary Γ, we get

$$\begin{aligned} & \Pi_{A_1:A_2}(\Pi_{A_2:A'_2}(\Pi^D_{A'_2:A_2}(\Pi_{A_2}(e_2))\Gamma_{g;A'_2=A_2;f}e_2)) \\ = \; & \Pi_{A_1:A'_2}(\Pi^D_{A'_2:A_2}(\Pi_{A_2}(e_2))\Gamma_{g;A'_2=A_2;f}e_2) \\ = \; & e_1\Gamma_{g;A_1=A_2;f}e_2. \end{aligned}$$

This is a special case of equivalence (4.27) for θ equal to $=$, so we know that the i-th tuple in rhs is

$$\begin{aligned} & t_i \circ [g : f(\sigma_{t_i|_{A_1}=A_2}(e_2))] \\ = \; & t_i \circ [g : f(\sigma_{A_1=A_2}(e_2))(t_i)]. \end{aligned}$$

A.3. Experimental Setup

In this section we provide details on the experimental setup used in our experiments.

Use case XMP:

```
<!DOCTYPE bib [
  <!ELEMENT bib       (book*)>
  <!ELEMENT book      ( title , (author+ |
                        editor +),
                        publisher ,
                        price )>
  <!ATTLIST book      year CDATA
                        #REQUIRED>
  <!ELEMENT author    ( last , first )>
  <!ELEMENT editor    ( last , first ,
                        affiliation )>
  <!ELEMENT title     (#PCDATA)>
  <!ELEMENT last      (#PCDATA)>
  <!ELEMENT first     (#PCDATA)>
  <!ELEMENT affiliation (#PCDATA)>
  <!ELEMENT publisher (#PCDATA)>
  <!ELEMENT price     (#PCDATA)>
]>

<!DOCTYPE reviews [
  <!ELEMENT reviews (entry*)>
  <!ELEMENT entry ( title , price ,
                    review )>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
  <!ELEMENT review (#PCDATA)>
]>

<!DOCTYPE prices [
  <!ELEMENT prices (book*)>
  <!ELEMENT book (title, source ,
                  price )>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT source (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>
```

Use case R:

```
<!DOCTYPE users [
  <!ELEMENT users    ( usertuple *)>
  <!ELEMENT usertuple (userid, name,
                       rating ?)>
  <!ELEMENT userid   (#PCDATA)>
  <!ELEMENT name     (#PCDATA)>
  <!ELEMENT rating   (#PCDATA)>
]>

<!DOCTYPE items [
  <!ELEMENT items     ( itemtuple *)>
  <!ELEMENT itemtuple (itemno,
                       description ,
                       offeredby ,
                       startdate ,
                       enddate ,
                       reserveprice )>
  <!ELEMENT itemno   (#PCDATA)>
  <!ELEMENT description (#PCDATA)>
  <!ELEMENT offered_by (#PCDATA)>
  <!ELEMENT startdate (#PCDATA)>
  <!ELEMENT enddate  (#PCDATA)>
  <!ELEMENT reserveprice (#PCDATA)>
]>

<!DOCTYPE bids [
  <!ELEMENT bids    ( bidtuple *)>
  <!ELEMENT bidtuple (userid, itemno,
                      bid , biddate )>
  <!ELEMENT userid (#PCDATA)>
  <!ELEMENT itemno (#PCDATA)>
  <!ELEMENT bid     (#PCDATA)>
  <!ELEMENT biddate (#PCDATA)>
]>
```

Figure A.1.: DTDs for the experimental data

A.3.1. System Setup

The system runs on an Intel Pentium 4 CPU 2.40GHz PC with 1 GB RAM and IBM 18.3GB Ultra 160 SCSI hard disk drive with 4MB buffer. Natix was compiled with GCC 3.3.5 and optimization level O3. All queries were run with warm buffer cache under Linux Kernel 2.6.11. The database buffer was 8 MB large.

A.3.2. Experimental Data

The data sets we used are based on the XQuery Use Cases "XMP" and "R". "XMP" contains data on books, reviews, prices and so on, while "R" describes an auction site with users, items, bids, etc.

The XML files were generated by ToXgene[1] using the DTD in the XQuery use case document. We repeat these DTDs in Figure A.1. To assess the scalability of each plan

[1] available at: http://www.cs.toronto.edu/tox/toxgene/

	Use case XMP			Use case R		
file	bib.xml	prices.xml	reviews.xml	bids.xml	items.xml	users.xml
100	68.7 KB	10.7 KB	20.8 KB	11.1 KB	21.4 KB	9.0 KB
1000	688 KB	106 KB	203 KB	111 KB	215 KB	89.4 KB
10000	6.90 MB	1.06 MB	2.07 MB	1.13 MB	2.16 MB	903 KB

Figure A.2.: File size of the input documents

alternative we executed the various evaluation plans on different sizes of input documents as listed in Figure A.2. The number of authors per book varied between 1 and 10. Similarly, there are between 1 and 10 bids per item. We note the number of elements contained in the input documents for each measurement and thereby reference to the documents as summarized in Figure A.2. We did not consider larger documents for evaluation because for most queries the nested queries did not even finish to evaluate on our largest document instances within three hours.

Bibliography

[AAC+01] Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors. *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases: Roma, Italy, 11–14th September, 2001*, San Mateo, CA, USA, 2001. Morgan Kaufmann.

[AAN01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In Apers et al. [AAC+01], pages 591–600.

[AC75] Morton M. Astrahan and Donald D. Chamberlin. Implementation of a Structured English Query Language. *Communications of the ACM*, 18(10):580–588, October 1975. Also published in/as: 19 ACM SIGMOD Conf. on the Management of Data, King(ed), May.1975.

[ACJK01] Michael O. Akinde, Damianos Chatziantoniou, Theodore Johnson, and Samuel Kim. The MD-join: An operator for complex OLAP. In *Proc. IEEE Conference on Data Engineering*, pages 524–533, 2001.

[ACM02] ACM, editor. *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS 2002: Madison, Wisconsin, June 3–5, 2002*, New York, NY 10036, USA, 2002. ACM Press. ACM order number 475021.

[Alb91] Joseph Albert. Algebraic properties of bag data types. In Rafael Camps, Guy M. Lohman, and Amílcar Sernadas, editors, *Proceedings. of the 17th Int. Conf. on Very Large Data Bases, September 3–6, 1991, Barcelona, Spain*, pages 211–219, San Mateo, CA, USA, 1991. Morgan Kaufmann.

[ALSU06] Alfred V Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[ALW+06] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. Cost-based query transformation in Oracle. In Dayal et al. [DWL+06], pages 1026–1036.

[Aly05] Robin Aly. Design and Implementation des Schema Managements für die XML Anfragesprachen XQuery und XPath. Master's thesis, University of Mannheim, 2005.

[Ano04] Anonymous, editor. *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*. IEEE Computer Society, 2004.

[Ano05] Anonymous, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, June 13-16, 2005*, New York, NY 10036, USA, 2005. ACM Press.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AYCLS01] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 497–508, June 2001.

[B+02] Philip A. Bernstein et al., editors. *VLDB 2002: proceedings of the Twenty-Eighth International Conference on Very Large Data Bases, Hong Kong SAR, China, 20–23 August 2002*, San Mateo, CA, USA, 2002. Morgan Kaufmann.

[BÖ5] Alexander Böhm. Integration of high-concurrency B^{LINK}-trees into Natix. Master's thesis, University of Mannheim, 2005.

[BBK+06] Alexander Böhm, Matthias Brantner, Carl-Christian Kanne, Norman May, and Guido Moerkotte. Natix visual interfaces. In *Proc. 10. International Conference on Extending Database Technology*, pages 1125–1129, 2006.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[BC04] V. Borkar and M. Carey. Extending XQuery for grouping, duplicate elimination, and outer joins. In *XML 2004*, November 2004.

[BCC+04] Kevin S. Beyer, Roberta Cochrane, Latha S. Colby, Fatma Ozcan, and Hamid Pirahesh. XQuery for analytics: Challenges and requirements. In *<XIME-P/>*, pages 3–8, 2004.

[BCC+05] Kevin Beyer, Don Chamberlin, Latha Colby, Fatma Özcan, Hamid Pirahesh, and Yu Xu. Extending XQuery for analytics. In Anonymous [Ano05], pages 503–515.

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. World Wide Web Consortion (W3C), January 2007. W3C Recommendation.

[BCH+06] Kevin S. Beyer, Roberta Cochrane, M. Hvizdos, Vanja Josifovski, Jim Kleewein, George Lapis, Guy M. Lohman, Robert Lyle, M. Nicola, Fatma Özcan, Hamid Pirahesh, Norman Seemann, Ashutosh Singh, Tuong C. Truong, Robbert C. Van der Linden, Brian Vickery, Chun Zhang, and G. Zhang. DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. *IBM Systems Journal*, 45(2):271–298, 2006.

[BCM05] Attila Barta, Mariano P. Consens, and Alberto O. Mendelzon. Benefits of path summaries in an XML query optimizer supporting multiple access methods. In Böhm et al. [BJH+05], pages 133–144.

[BCR+03] Zohra Bellahsene, Akmal B. Chaudhri, Erhard Rahm, Michael Rys, and Rainer Unland, editors. *Database and XML Technologies, First International XML Database Symposium, XSym 2003, Berlin, Germany, September 8, 2003, Proceedings*, volume 2824 of *Lecture Notes in Computer Science*. Springer, 2003.

[BD83] Dina Bitton and David J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems*, 8(2):255–265, June 1983.

[BEH+06] Andrey Balmin, Tom Eliaz, John Hornibrook, Lipyeow Lim, Guy M. Lohman, David E. Simmen, Min Wang, and Chun Zhang. Cost-based optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–320, 2006.

[BFS00] Peter Buneman, Mary F. Fernández, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases*, 9(1):76–110, 2000.

[BGI95] G. Bhargava, P. Goel, and B. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In Carey and Schneider [CS95], pages 304–315.

[BGvK+06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 479–490. ACM, 2006.

[Bit07] Susanne Bitzer. Design and implementation of a query unnesting module in Natix. Master's thesis, University of Mannheim, 2007.

[BJH+05] Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors. *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, New York, NY 10036, USA, 2005. ACM Press.

[BJZ94] Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors. *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, San Mateo, CA, USA, 1994. Morgan Kaufmann.

[BKHM05] Matthias Brantner, Carl-Christian Kanne, Sven Helmer, and Guido Moerkotte. Full-fledged algebraic XPath processing in Natix. In *Proc. IEEE Conference on Data Engineering*, pages 705–716, 2005.

[BKHM06] Matthias Brantner, Carl-Christian Kanne, Sven Helmer, and Guido Moerkotte. Algebraic optimization of nested XPath expressions. In Liu et al. [LRWZ06], page 128.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In Michael Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA*, pages 310–321, New York, NY 10036, USA, 2002. ACM Press. ACM order number 475020.

[BMM06] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting SQL queries in the presence of disjunction. Technical Report TR-2006-013, University of Mannheim, March 2006.

[BMM07] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting scalar SQL queries in the presence of disjunction. In *Proc. 23. ICDE Conference, Istanbul, Turkey*, 2007.

[BOB+04] Andrey Balmin, Fatma Ozcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A framework for using materialized XPath views in XML query processing. In Nascimento et al. [NÖK+04], pages 60–71.

[Bry89] François Bry. Towards an efficient evaluation of general queries: quantifier and disjunction processing revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 193–204, June 1989.

[BT99] Catriel Beeri and Yariv Tzaban. SAL: An algebra for semistructured data and XML. In *WebDB (Informal Proceedings)*, pages 37–42, 1999.

[CCF+06] Don Chamberlin, Mike Carey, Daniela Florescu, Donald Kossmann, and Jonathan Robie. Programming with XQuery. In Mike Carey and Hamid Pirahesh Torsten Grust, editors, *XIME-P*, 2006.

[CCM96] Vassilis Christophides, Sophie Cluet, and Guido Moerkotte. Evaluating queries with generalized path expressions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 413–422, 1996.

[CD92] Sophie Cluet and Claude Delobel. A general framework for the optimization of object-oriented queries. In Stonebraker [Sto92], pages 383–392.

[CD99] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. World Wide Web Consortion (W3C), November 1999. W3C Recomendation.

[CDF+86] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shekita. The architecture of the EXODUS extensible dbms. In Klaus R. Dittrich and Umeshwar Dayal, editors, *1986 International Workshop on Object-Oriented Database Systems, September 23-26, 1986, Asilomar Conference Center, Pacific Grove, California, USA, Proceedings*, pages 52–65. IEEE Computer Society, 1986.

[CDF+04] Don Chamberlin, Denise Draper, Mary F. Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jérôme Siméon, Jim Tivy, and Philip Wadler. *XQuery from the Experts – A Guide to the W3C XML Query Language*. Addison Wesley, 2004.

[CG85] Stefano Ceri and Georg Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, 1985.

[Che98] Mitch Cherniack. *Building Query Optimizers With Combinators*. PhD thesis, Brown University, December 1998.

[CKK98] Jens Claussen, Alfons Kemper, and Donald Kossmann. Order-preserving hash joins: Sorting (almost) for free. Technical Report MIP-9810, University of Passau, 1998.

[CKM+00] Jens Claussen, Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):238–260, 2000.

[CKM02] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic XML trees. In ACM [ACM02], pages 271–281. ACM order number 475021.

[CKMP97] Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In Jarke et al. [JCD+97], pages 286–295. Also known as VLDB'97.

[CM93] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pages 226–242. Springer, 1993.

[CM95a] S. Cluet and G. Moerkotte. Nested queries in object bases. Technical Report RWTH-95-06, GemoReport64, RWTH Aachen/INRIA, 1995.

[CM95b] Sophie Cluet and Guido Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[CM95c] Sophie Cluet and Guido Moerkotte. Efficient evaluation of aggregates on bulk types. In Paolo Atzeni and Val Tannen, editors, *Database Programming Languages (DBPL-5), Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995*, Electronic Workshops in Computing, page 8. Springer, 1995.

[CRF00] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. *Lecture Notes in Computer Science*, 1997:1–25, December 2000.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In Bocca et al. [BJZ94], pages 354–366.

[CS95] Michael J. Carey and Donovan A. Schneider, editors. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, New York, NY 10036, USA, 1995. ACM Press.

[CSF+01] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In Apers et al. [AAC+01], pages 341–350.

[CVZ+02] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In Bernstein et al. [B+02], pages 263–274.

[CZ96] Mitch Cherniack and Stanley B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 401–412, June 1996.

[CZ98] Mitch Cherniack and Stanley B. Zdonik. Changing the rules: Transformations for rule-based optimizers. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 61–72, 1998.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 197–208, San Mateo, CA, USA, 1987. Morgan Kaufmann.

[DB95] Dinesh Das and Don S. Batory. Praire: A rule specification framework for query optimizers. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 201–210. IEEE Computer Society, 1995.

[DFF+98] Alin Deutsch, Mary F. Fernández, Daniela Florescu, Alon Y. Levy, and Dan Suciu. XML-QL: A query language for XML. In *WWW The Query Language Workshop (QL)*, Cambridge, MA, December 1998. World Wide Web Consortion (W3C).

[DFF+07] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. World Wide Web Consortion (W3C), January 2007. W3C Recommendation.

[DGK82] Umeshwar Dayal, Nathan Goodman, and Randy H. Katz. An extended relational algebra with control over duplicate elimination. In *Proc. ACM SIGMOD/SIGACT Conf. on Principles of Database Systems. (PODS)*, pages 117–123, 1982.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 14(2):1–8, 1984.

[DPX04] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT logical framework for XQuery. In Nascimento et al. [NÖK+04], pages 168–179.

[DT87] Umeshwar Dayal and Irv Traiger, editors. *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data 1987 annual conference, San Francisco, May 27–29, 1987*, New York, NY 10036, USA, 1987. ACM Press. ACM order number 472870.

[DWL+06] Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors. *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 2006.

[EGLGJ07] Mostafa Elhemali, César Galindo-Legaria, Thorsen Grabs, and Milind Joshi. Execution strategies for SQL subqueries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 993–1004, June 2007.

[Eng07] Daniel Engovatov. *XML Query 1.1 Requirements*. World Wide Web Consortion (W3C), March 2007. W3C Working Draft.

[Feg98] Leonidas Fegaras. Query unnesting in object-oriented databases. In Laura Haas and Ashutosh Tiwary, editors, *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data: June 1–4, 1998, Seattle, Washington, USA*, volume 27(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 49–60, New York, NY 10036, USA, 1998. ACM Press.

[FHK+02] 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: Very Large Data Bases*, 11(4):292–314, December 2002.

[FHK+04] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, and Arvind Sundararajan. The BEA streaming XQuery processor. *VLDB Journal: Very Large Data Bases*, 13(3):294–315, 2004.

[FHM+04] Mary Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Automata for avoiding unnecessary ordering operations in XPath implementations. Technical report TR UA 2004-02, University of Antwerp, 2004.

[FHP02] Flavius Frasincar, Geert-Jan Houben, and Cristian Pau. XAL: An algebra for XML query optimization. In Xiaofang Zhou, editor, *Thirteenth Australasian Database Conference (ADC2002)*, Melbourne, Australia, 2002. ACS.

[FHR+02] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. StatiX: making XML count. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 181–191, New York, NY 10036, USA, 2002. ACM Press.

[FK04] Daniela Florescu and Donald Kossmann. XML query processing (tutorial). In Anonymous [Ano04].

[FKS+02] John E. Funderburk, Gerald Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Catalina Wei. XTABLES: Bridging relational technology and XML. *IBM Systems Journal*, 41(4):616–641, November 2002.

[FLA+03] Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors. *Very large data bases: 2003, 29th VLDB, Berlin, Germany, September 9–12: Proceedings of 29th International Conference on Very Large Data Bases*, San Mateo, CA, USA, 2003. Morgan Kaufmann.

[FLBC02] Leonidas Fegaras, David Levine, Sujoe Bose, and Vamsi Chaluvadi. Query processing of streamed XML data. In *Proc. CIKM*, pages 126–133, New York, NY 10036, USA, 2002. ACM Press.

[FM95] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In Carey and Schneider [CS95], pages 47–58.

[FM00] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000.

[FM01] Thorsten Fiebig and Guido Moerkotte. Algebraic XML construction and its optimization in Natix. *WWW Journal*, 4(3):167–187, 2001.

[FMS93] Leonidas Fegaras, David Maier, and Tim Sheard. Specifying rule-based query optimizers in a reflective framework. In *Third International Conference on Deductive and Object-Oriented Databases*, pages 146–168, Phoenix, Arizona, 1993.

[Fre87] Johann Christoph Freytag. A rule-based view of query optimization. In Dayal and Traiger [DT87], pages 173–180. ACM order number 472870.

[FS98] Mary F. Fernández and Dan Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 14–23. IEEE Computer Society, 1998.

[GC95] Goetz Graefe and Richard L. Cole. Fast algorithms for universal quantification in large databases. *ACM Transactions on Database Systems*, 20(2):187–236, June 1995.

[GCD+94] Goetz Graefe, Richard L. Cole, Diane L. Davison, William J. McKenna, and Richard H. Wolniewicz. Extensible query optimization and parallel execution in Volcano. In Johann Christoph Freytag, David Maier, and G. Vossen, editors, *Query Processing for Advanced Database Systems, (Dagstuhl, June 1991)*, pages 305–335. Kaufmann, 1994.

[GD87] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In Dayal and Traiger [DT87], pages 160–172. ACM order number 472870.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements Of Reusable Object Oriented Software*. Addison Wesley Longman, 1995.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In Bernstein et al. [B$^+$02], pages 95–106.

[GKP03a] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of XPath query evaluation. In *Proc. ACM SIGMOD/SIGACT Conf. on Principles of Database Systems. (PODS)*, San Diego, California, USA, 2003. ACM Press.

[GKP03b] Georg Gottlob, Christoph Koch, and Reinhard Pichler. XPath query evaluation: Improving time and space efficiency. In *Proc. IEEE Conference on Data Engineering*, pages 379–390, Bangalore, India, 2003. IEEE Computer Society.

[GKP05] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005.

[GLJ01] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):571–581, June 2001.

[GLR97] César Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–73, March 1997.

[GLS94] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. Sort vs. hash revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):934–944, December 1994.

[GM93] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. *IEEE Trans. On Data Eng.*, pages 209–218, April 1993.

[GMUW01] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 1st edition, 2001.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra94] Goetz Graefe. Sort-merge-join: An idea whose time has(h) passed? In IEEE [IEE94], pages 406–417.

[Gra95] Goetz Graefe. The cascades framework for query optimization. *Bulletin of the Technical Committee on Data Engineering*, 18(3):19–28, September 1995.

[Gra03] Goetz Graefe. Executing nested queries. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *BTW*, volume 26 of *LNI*, pages 58–77. GI, 2003.

[GRS05] Ravindra Guravannavar, H. S. Ramanujam, and S. Sudarshan. Optimizing nested queries with parameter sort orders. In Böhm et al. [BJH$^+$05], pages 481–492.

[GRT07] Torsten Grust, Jan Rittinger, and Jens Teubner. eXrQuy: Order indifference in XQuery. In *Proceedings of the 23rd IEEE Int'l Conference on Data Engineering (ICDE 2007)*, pages 226–235. IEEE, 2007.

[Gru02] Torsten Grust. Accelerating XPath location steps. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 109–120, June 2002.

[GST04] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL hosts. In Nascimento et al. [NÖK+04], pages 252–263.

[GT04] Torsten Grust and Jens Teubner. Relational algebra: Mother tongue - XQuery: Fluent. In *Twente Data Management Workshop on XML Databases and Information Retrieval (TDM) 04, (informal proceedings)*, 2004.

[GvKT03] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In Freytag et al. [FLA+03], pages 524–525.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of nested SQL queries revisited. In Dayal and Traiger [DT87], pages 23–33. ACM order number 472870.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In Jarke et al. [JCD+97], pages 436–445. Also known as VLDB'97.

[Hac06] Georg Hackenberg. An extensible property module for a plan generator. Bachelor's thesis, University of Mannheim, 2006.

[HCLS97] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. Seeking the truth about *ad hoc* join costs. *VLDB Journal: Very Large Data Bases*, 6(3):241–256, May 1997. Electronic edition.

[HDN+03] Alan Halverson, David DeWitt, Jeffrey Naughton, Feng Tian, Yuan Wang, Josef Burger, Rajasekar Krishnamurthy, Stratis Viglas, Ameet Kini, and Ajith Nagaraja Rao. Mixed mode XML query processing. In Freytag et al. [FLA+03], pages 225–236.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensive query processing in Starburst. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 18(2):377–388, June 1989.

[Hid03] Jan Hidders. Satisfiability of XPath expressions. In Lausen and Suciu [LS04], pages 21–36.

[HKM02] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Optimized translation of XPath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *Proceedings of the 3nd International Conference on Web Information Systems Engineering (WISE'02)*, pages 215–224. IEEE Computer Society, 2002.

[HM97] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 386–395, 1997.

[HM03] Jan Hidders and Philippe Michiels. Avoiding unnecessary ordering operations in XPath. In Lausen and Suciu [LS04], pages 54–74.

[HNM02] Sven Helmer, Thomas Neumann, and Guido Moerkotte. Early grouping gets the skew. Technical report, University of Mannheim, 2002.

[HNM03] Sven Helmer, Thomas Neumann, and Guido Moerkotte. Estimating the output cardinality of partial preaggregation with a measure of clusteredness. In Freytag et al. [FLA+03], pages 656–667.

[HPVD04] Jan Hidders, Jan Paredaens, Roel Vercammen, and Serge Demeyer. A light but formal introduction to XQuery. In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, *XSym*, volume 3186 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2004.

[Hun06] Jason Hunter. Web publishing 2.0. In *XML 2006*, December 2006.

[IC91] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 268–277, 1991.

[IEE94] IEEE, editor. *Proc. of the 10th International Conference On Data Engineering, Houston, TX*, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[ILW03] Zachary G. Ives, Alon Y. Levy, and Daniel S. Weld. Efficient evaluation of regular path expressions on streaming xml data. Technical Report UW-CSE-2000-05-02, University of Washington, 2003.

[Ioa03] Yannis E. Ioannidis. The history of histograms (abridged). In Freytag et al. [FLA+03], pages 19–30.

[JAKC+02] 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: Very Large Data Bases*, 11(4):274–291, December 2002.

[JCD+97] Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors. *Proceedings of the Twenty-third International Conference on Very Large Data Bases, Athens, Greece, 26–29 August 1997*, San Mateo, CA, USA, 1997. Morgan Kaufmann. Also known as VLDB'97.

[JFB05] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *VLDB Journal: Very Large Data Bases*, 14(2):197–210, 2005.

[JHSV06] Philippe Michiels Jan Hidders, Jérôme Siméon, and Roel Vercammen. How to recognize different kinds of tree patterns from quite a long way away. Technical Report TR UA 13-2006, University of Antwerp, 2006.

[JJP+02] Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors. *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, volume 2287 of *Lecture Notes in Computer Science*. Springer, 2002.

[JK84] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[JLST02] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A tree algebra for XML. In *DBPL*, pages 149–164, 2002.

[JM96] H. V. Jagadish and Inderpal Singh Mumick, editors. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, New York, NY 10036, USA, 1996. ACM Press.

[JSSS05] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency control and recovery for balanced B-link trees. *VLDB Journal: Very Large Data Bases*, 14(2):257–277, 2005.

[Kan02] Carl-Christian Kanne. *Core technologies for Native XML database management systems*. PhD thesis, University of Mannheim, 2002.

[Kay07] Michael Kay. Saxon XSLT and XQuery Processor. http://www.saxonia.com, 2007.

[KBM05] Carl-Christian Kanne, Matthias Brantner, and Guido Moerkotte. Cost-sensitive reordering of navigational primitives. In Anonymous [Ano05], pages 742–753.

[KD99] Navin Kabra and David J. DeWitt. OPT++ : an object-oriented implementation for extensible database query optimization. *VLDB Journal: Very Large Data Bases*, 8(1):55–78, May 1999. Electronic edition.

[KG02] April Kwong and Michael Gertz. Schema-based optimization of XPath expressions. Technical report, University of California Davis, 2002.

[Kie84] W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.

[Kim82] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[KKN03] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F. Naughton. XML-SQL query translation literature: The state of the art and open problems. In Bellahsene et al. [BCR+03], pages 1–18.

[Klu82] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, July 1982.

[KM90] Alfons Kemper and Guido Moerkotte. Access support in object bases. In Anonymous, editor, *Proc. of the ACM SIGMOD Conf. on Management of Data*, volume 19(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 364–374, New York, NY 10036, USA, June 1990. ACM Press.

[KM00] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML data. In *Proc. IEEE Conference on Data Engineering*, page 198, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. IEEE Computer Society Press.

[KM06] Carl-Christian Kanne and Guido Moerkotte. A linear time algorithm for optimal tree sibling partitioning and approximation algorithms in Natix. In Dayal et al. [DWL+06], pages 91–102.

[Lar02] Per-Åke Larson. Data reduction by partial preaggregation. In *Proc. IEEE Conference on Data Engineering*, pages 706–715. IEEE Computer Society Press, 2002.

[LFL88] Mavis K. Lee, Johann Christoph Freytag, and Guy M. Lohman. Implementing an interpreter for functional rules in a query optimizer. In François Bancilhon and David J. DeWitt, editors, *Very large data bases: 1988, 14th VLDB,*

Los Angeles, USA, August 29–September 1: proceedings of the Fourteenth International Conference on Very Large Data Bases, pages 218–229, San Mateo, CA, USA, 1988. Morgan Kaufmann. Sponsored by VLDB Endowment, ACM-SIGMOD, and the Computer Society of the IEEE.

[LKA05] Zhen Hua Liu, Muralidhar Krishnaprasad, and Vikas Arora. Native XQuery processing in Oracle XML DB. In Anonymous [Ano05], pages 828–833.

[LM01] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In Apers et al. [AAC+01], pages 361–370.

[LMP02] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A transducer-based XML query processor. In Bernstein et al. [B+02], pages 227–238.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):18–27, September 1988.

[LRWZ06] Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors. *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. IEEE Computer Society, 2006.

[LS03] Alberto Lerner and Dennis Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In Freytag et al. [FLA+03], pages 345–356.

[LS04] Georg Lausen and Dan Suciu, editors. *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*, volume 2921 of *Lecture Notes in Computer Science*. Springer, 2004.

[LWP+02] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ronald Parr. XPathLearner: An on-line self-tuning Markov histogram for XML path selectivity estimation. In Bernstein et al. [B+02], pages 442–453.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1983.

[May02] Norman May. Design und Implementierung eines adaptierbaren Codegenerators für Datenbanksysteme. Master's thesis, University of Mannheim, July 2002.

[MBB+06] Norman May, Matthias Brantner, Alexander Böhm, Carl-Christian Kanne, and Guido Moerkotte. Index vs. navigation in XPath evaluation. In *Proc. 4th Int. XML Database Symposium (XSym 2006), Seoul, Korea*, pages 16–30, 2006.

[MBV03] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML web: a first study. In *WWW*, pages 500–510, 2003.

[McK93] William J. McKenna. *Efficient Search in Extensible Databse Query Optimization: The Volcano Optimizer Generator*. PhD thesis, University Of Colorado at Boulder, 1993.

[MCS88] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, September 1988.

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[MFK01] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries on heterogeneous data sources. In Apers et al. [AAC+01], pages 241–250.

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):247–258, June 1990.

[MGM03] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems*, 28(1):56–99, 2003.

[MHKM04] Norman May, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. XQuery processing in Natix with an emphasis on join ordering. In Ioana Manolescu and Yannis Papakonstantinou, editors, *XIME-P*, pages 49–54, June 2004.

[MHM03a] Norman May, Sven Helmer, and Guido Moerkotte. Nested queries and quantifiers in an ordered context. Technical Report TR-03-002, Department for Mathematics and Computer Science, University of Mannheim, 2003.

[MHM03b] Norman May, Sven Helmer, and Guido Moerkotte. Quantifiers in XQuery. In *Proc. 4th Int. Conf. on Web Information Systems Engineering (WISE), Rome, Italy*, pages 313–316, 2003.

[MHM03c] Norman May, Sven Helmer, and Guido Moerkotte. Three cases for query decorrelation in XQuery. In Bellahsene et al. [BCR+03], pages 70–84.

[MHM04] Norman May, Sven Helmer, and Guido Moerkotte. Nested queries and quantifiers in an ordered context. In Anonymous [Ano04], pages 239–250.

[MHM06] Norman May, Sven Helmer, and Guido Moerkotte. Strategies for query unnesting in XML databases. *ACM Transactions on Database Systems*, 31(3):968–1013, September 2006.

[MM05a] Norman May and Guido Moerkotte. Main memory implementations for binary grouping. In Stéphane Bressan, Stefano Ceri, Ela Hunt, Zachary G. Ives, Zohra Bellahsene, Michael Rys, and Rainer Unland, editors, *Proc. 3rd Int. XML Database Symposium (XSym 2005), Trondheim, Norway*, volume 3671 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2005.

[MM05b] Norman May and Guido Moerkotte. Main memory implementations for binary grouping. Technical Report TR-05-007, University of Mannheim, 2005.

[MMS06] P. Michiels, G. A. Mihăilă, and J. Siméon. Put a tree pattern in your algebra. Technical report, Univ. of Antwerp, TR-06-09, Belgium, 2006.

[Moe03] Guido Moerkotte. Constructing optimal bushy trees possibly containing cross products for order-preserving joins is in P. Technical Report MA-03-12, University of Mannheim, 2003.

[Moe05] Guido Moerkotte. Building query compilers. unpublished, 2005.

[Moe06] Guido Moerkotte. Dp-counter analytics. Technical Report TR-2006-002, University of Mannheim, 2006.

[MP94] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):103–114, June 1994.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT '99: Proceeding of the 7th International Conference on Database Theory*, pages 277–295, London, UK, 1999. Springer-Verlag.

[MS02] Gerome Miklau and Dan Suciu. Containment and equivalence for an XPath fragment. In ACM [ACM02], pages 65–76. ACM order number 475021.

[Mur89] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In P. M. G. (Petrus Maria Gerardus) Apers and Gio Wiederhold, editors, *Very large data bases: proceedings: proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22–25, 1989, Amsterdam, The Netherlands*, pages 77–85, San Mateo, CA, USA, 1989. Morgan Kaufmann.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In Li-Yan Yuan, editor, *Very large data bases: VLDB '92, proceedings of the 18th International Conference on Very Large Data Bases, August 23–27, 1992, Vancouver, Canada*, pages 91–102, San Mateo, CA, USA, 1992. Morgan Kaufmann.

[MW99] Jason McHugh and Jennifer Widom. Query optimization for XML. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 315–326. Morgan Kaufmann, 1999.

[Nak90] Ryohei Nakano. Translation with optimization from relational calculus to relational algebra having aggregate functions. *ACM Transactions on Database Systems*, 15(4):518–557, December 1990.

[NdL05] Matthias Nicola and Bert Van der Linden. Native XML support in DB2 universal database. In Böhm et al. [BJH+05], pages 1164–1174.

[NDM+01] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara internet query system. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[Neu01] Thomas Neumann. Regelbasierte Plangenerierung. Master's thesis, University of Mannheim, 2001.

[Neu05] Thomas Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. PhD thesis, University of Mannheim, 2005.

[NM04] Thomas Neumann and Guido Moerkotte. A combined framework for grouping and order optimization. In Nascimento et al. [NÖK+04], pages 960–971.

[NM06] Thomas Neumann and Guido Moerkotte. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In Dayal et al. [DWL+06].

[NÖK+04] Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors. *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*. Morgan Kaufmann, 2004.

[NPS91] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Transactions on Database Systems*, 16(3):513–534, 1991.

[OCP+05] Fatma Ozcan, Roberta Cochrane, Hamid Pirahesh, Jim Kleewein, Kevin S. Beyer, Vanja Josifovski, and Chun Zhang. System RX: One part relational, one part XML. In Anonymous [Ano05], pages 347–358.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In Dennis McLeod, Ron Sacks-Davis, and H.-J Schek, editors, *Very large data bases: 16th International Conference on Very Large Data Bases; August 13–16, 1990, Brisbane, Australia*, pages 314–325, San Mateo, CA, USA, 1990. Morgan Kaufmann.

[OMFB02] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking forward. *Lecture Notes in Computer Science*, 2490:109–127, 2002.

[OOP+04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-friendly XML node labels. In Weikum et al. [WKD04], pages 903–908.

[PAKJ+02] Stelios Paparizos, Shurug Al-Khalifa, H.V. Jagadish, Laks Lakshmanan, Andrew Nierman, Divesh Srivastava, and Yuqing Wu. Grouping in XML. *XMLDM'02, Lecture Notes in Computer Science*, 2490:128–147, 2002.

[PCS+05] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, and Eugene Kogan. XQuery implementation in a relational database system. In Böhm et al. [BJH+05], pages 1175–1186.

[PG02] Neoklis Polyzotis and Minos Garofalakis. Statistical synopses for graph-structured XML databases. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 358–369, New York, NY 10036, USA, 2002. ACM Press.

[PG06] Neoklis Polyzotis and Minos Garofalakis. Xcluster synopses for structured xml content. In Liu et al. [LRWZ06], page 63.

[PGI04] Neoklis Polyzotis, Minos N. Garofalakis, and Yannis E. Ioannidis. Approximate XML query answers. In Weikum et al. [WKD04], pages 263–274.

[PGLK97] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten. The complexity of transformation-based join enumeration. In Jarke et al. [JCD+97], pages 306–315. Also known as VLDB'97.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 251–260. IEEE Computer Society, 1995.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. In Stonebraker [Sto92], pages 39–48.

[PHIS96] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In Jagadish and Mumick [JM96], pages 294–305.

[PLH97] Hamid Pirahesh, T. Y. Cliff Leung, and Waqar Hasan. A rule engine for query transformation in Starburst and IBM DB2 C/S DBMS. In Alex Gray and Per-Åke Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 391–400. IEEE Computer Society, 1997.

[PMC02] Chang-Won Park, Jun-Ki Min, and Chin-Wan Chung. Structural function inlining technique for structurally recursive XML queries. In *28th International Conference on Very Large Data Bases, August 20-23, 2002 Hong Kong, China*, August 2002.

[Ram06] Maya Ramanath. *Schema-based Statistics and Storage for XML*. PhD thesis, SERC, Indian Institute of Science, Bangalore, India, April 2006.

[RG02] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2002.

[RGL90] Arnon Rosenthal and César Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 291–299, New York, NY, USA, 1990. ACM Press.

[RKS88] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988. See comment [TG92].

[RLL+01] Jun Rao, Bruce Lindsay, Guy Lohman, Hamid Pirahesh, and David Simmen. Using EELs, a practical approach to outerjoin and antijoin reordering. In *Proc. IEEE Conference on Data Engineering*, page 0585, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[RM06] Ralf Rantzau and Christoph Mangold. Laws for rewriting queries containing division operators. In Liu et al. [LRWZ06], page 21.

[RPNR00] K. Ramasamy, J.M. Patel, J.F. Naughton, and R.Kaushik. Set containment joins: The good, the bad, and the ugly. In *Proc. of the 26th VLDB Conference*, Cairo, Egypt, August 2000.

[RSF06] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A complete and efficient algebraic compiler for XQuery. In Liu et al. [LRWZ06], page 14.

[RSMW02] Ralf Rantzau, Leonard D. Shapiro, Bernhard Mitschang, and Quan Wang. Universal quantification in relational databases: A classification of data and algorithms. In Jensen et al. [JJP+02], pages 445–463.

[SABdB94] Hennie J. Steenhagen, Peter M. G. Apers, Henk M. Blanken, and Rolf A. de By. From nested-loop to join queries in OODB. In Bocca et al. [BJZ94], pages 618–629.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of ACM SIGMOD Conference*, New York, NY 10036, USA, 1979. ACM Press.

[SAKJ+02] Divesh Srivastava, Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. IEEE Conference on Data Engineering*, pages 141–152, 2002.

[Sar03a] Carlo Sartiani. Evaluating nested queries on XML data. In *7th International Database Engineering and Applications Symposium (IDEAS 2003), 16-18 July 2003, Hong Kong, China*, pages 224–229, 2003.

[Sar03b] Carlo Sartiani. A general framework for estimating XML query cardinality. In Lausen and Suciu [LS04], pages 257–277.

[SB98] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[Sch01] Harald Schöning. Tamino - a DBMS designed for XML. In *Proc. IEEE Conference on Data Engineering*, pages 149–154, 2001.

[Sch04] Thomas Schwentick. XPath query containment. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 33(1):101–109, 2004.

[Seg03] Luc Segoufin. Typing and querying XML documents: Some complexity bounds. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2003)*, San Diego, California, USA, 2003. ACM Press.

[SHP+96] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakirshnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):435–446, 1996.

[SJS02] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Bringing order to query optimization. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 31(2):5–14, 2002.

[SKS+01] Jayavel Shanmugasundaram, Gerald Kiernan, Eugene J. Shekita, Catalina Fan, and John E. Funderburk. Querying XML views of relational data. In Apers et al. [AAC+01], pages 261–270.

[SPL96] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex query decorrelation. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 450–458. IEEE Computer Society, 1996.

[SSKH+02] S.Paparizos, S.Al-Khalifa, H.V.Jagadish, A.Nierman, and Y.Wu. A physical algebra for XML. Technical report, Database Group at the University of Michigan, 2002.

[SSM96] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 57–67, 1996.

[Ste95] Hennie J. Steenhagen. *Optimization of Object Query Languages*. PhD thesis, University of Twente, Enschede, The Netherlands, 1995. Translation from calculus to algebra, normalization, query rewriting, query unnesting.

[Sto92] Michael Stonebraker, editor. *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–5, 1992*, volume 21(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, New York, NY 10036, USA, 1992. ACM Press.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Suc01] Dan Suciu. On database theory and XML. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(3):39–45, September 2001.

[Suc02] Dan Suciu. The XML typechecking problem. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 31(1):89–96, March 2002.

[TDCZ02] Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 31(1):5–10, 2002.

[Tec07] Data Direct Technologies. Data Direct XQuery Processor. http://www.datadirect.com/products/xquery, 2007.

[TG92] Abdullah U. Tansel and Lucy Garnett. On M. A. Roth, H. F. Korth and A. Silberschatz: "Extended Algebra and Calculus for Nested Relational Databases" [ACM Trans. Database Systems **13** (1988), no. 4, 389–417]. *ACM Transactions on Database Systems*, 17(2):374–383, June 1992. See [RKS88].

[Van98] Bennet Vance. *Join-order optimization with cartesian products*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1998.

[vB87] Günter von Bültzingsloewen. Translating and optimizing SQL queries having aggregates. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 235–243, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers.

[vB90] G. von Bültzingsloewen. *Optimierung von SQL-Anfragen für parallele Bearbeitung (Optimization of SQL-queries for parallel processing)*. PhD thesis, University of Karlsruhe, 1990. in German.

[VGD+02] Stratis D. Viglas, Leonidas Galanis, David J. DeWitt, David Maier, and Jeffrey F. Naughton. Putting XML query algebras into context. Technical report, University of Wisconsin, Madison, Computer Science Department, 2002.

[VM96] Bennet Vance and David Maier. Rapid bushy join-order optimization with Cartesian products. In Jagadish and Mumick [JM96], pages 35–46.

[VR81] Y. L. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24(1):83–84, February 1981.

[WC03] Xiaoyu Wang and Mitch Cherniack. Avoiding ordering and grouping in query processing. In Freytag et al. [FLA+03], pages 826–837.

[Wei06] Felix Weigel. *Structural Summaries as a Core Technology for Efficient XML Retrieval*. PhD thesis, Ludwig-Maximilians-Universitt Mnchen, December 2006.

[Wes00] Till Westmann. *Effiziente Laufzeitsysteme für Datenlager*. PhD thesis, University of Mannheim, Germany, 2000.

[WJLY04] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Bloom histogram: Path selectivity estimation for XML data with updates. In Nascimento et al. [NÖK+04], pages 240–251.

[WKD04] Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, New York, NY 10036, USA, 2004. ACM Press.

[WM99] Till Westmann and Guido Moerkotte. Variations on grouping and aggregation. Technical Report MA-99-11, University of Mannheim, 1999.

[Woo01] Peter T. Wood. Minimising simple XPath expressions. In *WebDB*, pages 13–18, 2001.

[WPJ02] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In Jensen et al. [JJP+02], pages 590–608.

[WPJ03] Yuqing Wu, Jignesh Patel, and H.V. Jagadish. Structural join order selection for XML query optimization. In *Proc. IEEE Conference on Data Engineering*, March 2003.

[WY76] Eugene Wong and Karel Youssefi. Decomposition a strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223–241, 1976.

[YL94] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In IEEE [IEE94], pages 89–100.

[ZHJ+05] Ning Zhang, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. Statistical learning techniques for costing XML queries. In Böhm et al. [BJH+05], pages 289–300.

[ZND+01] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 425–436, June 2001.

[ZO02] Ning Zhang and Tamer Ozsu. Optimizing correlated path queries in XML languages. Technical Report CS-2002-36, School of Computer Science, University of Waterloo, November 2002.

[ZOAI06] Ning Zhang, Tamer Özsu, Ashraf Aboulnaga, and Ihab F. Ilyas. XSeed: Accurate and fast cardinality estimation for XPath queries. In Liu et al. [LRWZ06], page 61.